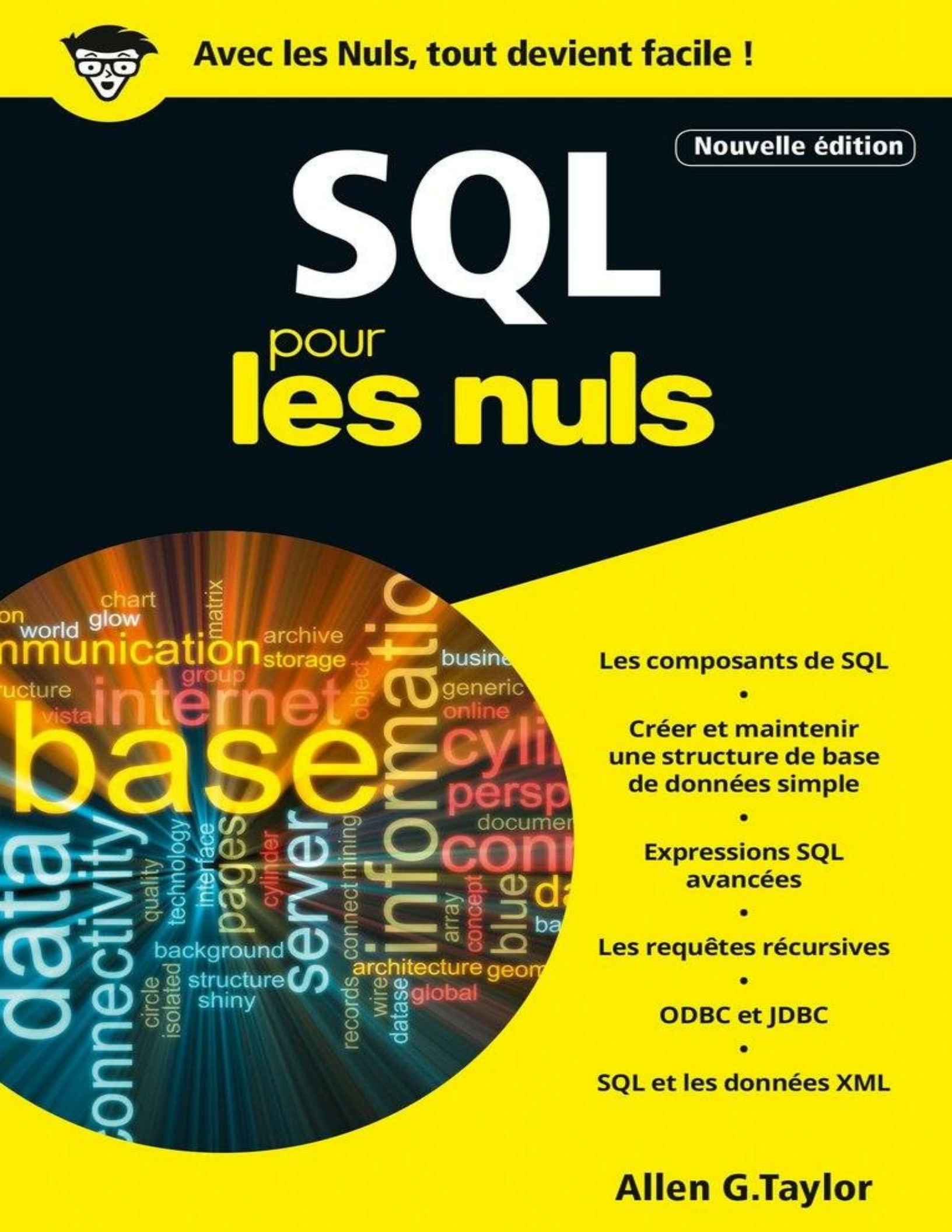

Avec les Nuls, tout devient facile !
Nouvelle édition
SQL
pour
les nuls
Les composants de SQL
•
Créer et maintenir une structure de base de données simple
•
Expressions SQL avancées
•
Les requêtes récursives
•
ODBC et JDBC
•
SQL et les données XML
Allen G.Taylor

SQL

pour les nuls

Nouvelle édition

Allen G. Taylor

FIRST Editions

SQL pour les Nuls (nouvelle édition)

Titre de l'édition originale : *SQL For Dummies®, 8th Edition*

Collection dirigée par Jean-Pierre Cano
Traduction : Jean-Paul Duplan et Stéphane Bontemps
Mise en page : maged

Edition française publiée en accord avec Wiley Publishing, Inc.

Éditions First, un département d'Édi8
12 avenue d'Italie
75013 Paris
Tél. : 01 44 16 09 00
Fax : 01 44 16 09 01
E-mail : firstinfo@efirst.com
Web : www.editionsfirst.fr
ISBN : 978-2-412-02377-8
ISBN numérique : 9782412026458
Dépôt légal : 1er trimestre 2017

Ce livre numérique a été converti initialement au format EPUB par Isako
www.isako.com à partir de l'édition papier du même ouvrage.

Introduction

Bienvenue dans le monde des bases de données et de son langage SQL (langage structuré de requêtes). Il existe de nombreux systèmes de gestion de bases de données (SGBD) pour de nombreuses plates-formes matérielles. Ces produits peuvent parfois être très différents, mais tous ceux qui sont dignes de ce nom partagent quelque chose en commun : ils permettent l'accès et la manipulation des données via SQL. Connaître SQL, c'est pouvoir créer des bases de données relationnelles et en extraire n'importe quelle information.

Au sujet de ce livre

Les systèmes de gestion de base de données sont au cœur de nombreuses organisations. Les gens pensent souvent que la création et l'administration de ces systèmes sont des tâches très complexes, réservées à des technogourous spécialisés en bases de données qui disposent de connaissances que le simple mortel ne pourrait espérer acquérir. Ce livre balaie ce mythe. Voici ce que vous y trouverez :

- Découvrir les notions élémentaires des bases de données.
- Comprendre comment un SGBD est structuré.
- Découvrir les fonctionnalités majeures de SQL.
- Créer une base de données.
- Protéger une base de données.
- Travailler sur les données d'une base de données.
- Déterminer comment extraire l'information que vous souhaitez d'une base de données.

L'objectif de ce livre est de vous apprendre à créer des bases de données relationnelles et à en extraire des informations utiles à l'aide de SQL. SQL est un langage standard international utilisé dans le monde entier pour créer et administrer des bases de données. Cette édition traite de la dernière version du standard, SQL:2011.

Ce livre ne vous apprendra pas comment produire un modèle de données. Je suppose que vous-même ou quelqu'un d'autre l'auront déjà fait. Cependant, cet ouvrage vous montrera comment implémenter ce modèle à l'aide de SQL. Si vous pensez que votre modèle de données n'est pas bon, vous devez absolument le repenser avant de construire la base de données correspondante. Plus tôt vous détecterez et corrigerez ce type de problème de conception, moins il vous en coûtera.

Qui devrait lire ce livre ?

Connaître SQL vous rendra service chaque fois que vous devrez stocker ou récupérer des données dans un SGBD. Vous n'avez pas besoin d'être programmeur pour utiliser SQL, pas plus que vous n'avez à connaître des langages tels que le COBOL, le C ou le Basic. La syntaxe de SQL est entièrement en anglais (et non en français, désolé).

Si vous êtes programmeur, vous pourrez incorporer SQL dans vos programmes pour manipuler plus facilement des données. Ce livre vous montrera comment procéder afin d'intégrer dans vos applications les riches et puissants outils qu'offre ce langage.

Icônes utilisées dans ce livre

Ce conseil vous permet de gagner du temps et de lever des doutes.

Souvenez-vous de l'information signalée par cette icône car vous pourrez en avoir besoin ultérieurement.

Tenez compte du conseil qui Figure à côté de cette icône. Il peut vous garantir contre des problèmes majeurs. Ignorez-le à vos risques et périls.

Cette icône vous signale la présence de détails techniques qu'il est intéressant mais pas indispensable de connaître.

Pour commencer

Les bases de données sont les meilleurs outils qui aient été inventés pour conserver la trace des données qui vous importent. Comprendre comment elles fonctionnent et maîtriser le langage SQL feront de vous un homme ou une femme en vogue. Vos collègues viendront vous voir dès lors qu'ils ou elles auront besoin d'informations essentielles. Vos chefs vous solliciteront pour des conseils. Les plus jeunes vous demanderont des autographes. Mais plus que tout, vous pourrez comprendre comment fonctionne votre entreprise (ou votre association) dans ses moindres détails.

Débuter en SQL

DANS CETTE PARTIE :

Les fondements des bases de données relationnelles.

Les concepts de base de SQL.

Les outils essentiels des bases de données.

Chapitre 1
Les bases de données relationnelles

DANS CE CHAPITRE :

- **Organiser l'information.**
- **Définir la notion de bases de données.**
- **Définir ce qu'est un SGBD.**
- **Comparer différents modèles de bases de données.**
- **Définir le concept de base de données relationnelle.**
- **Traiter des problèmes de conception d'une base de données.**

SQL (Structured Query Language, ou langage structuré de requêtes – prononcez *esse-q-elle* et non *squale*) est un langage standard spécifiquement conçu pour permettre aux gens de créer des bases de données, d'y ajouter de nouvelles données, d'assurer la maintenance de ces données et d'en récupérer des portions précises. Développé dans les années 1970 par IBM, SQL a évolué au fil des ans jusqu'à devenir un standard industriel. En tant que tel, il est maintenu par l'organisation internationale des standards (ISO).

Il existe plusieurs types de bases de données dont chacun correspond à un concept de modélisation particulier.

SQL a été initialement développé pour travailler sur des données contenues dans des bases bâties selon le modèle relationnel. Récemment, le standard SQL a évolué pour incorporer des éléments du modèle objet, produisant ainsi des structures hybrides appelées bases de données objet-relationnel (ou relationnel-objet, selon les auteurs). Dans ce chapitre, je traite du stockage de données, je consacre une section à expliquer comment le

modèle relationnel se distingue des autres modèles les plus connus, et je présente les fonctionnalités les plus importantes des bases de données relationnelles.

Avant de parler de SQL, je voudrais préciser un point : je dois expliquer ce que j'entends par base de données. La signification de ce terme a évolué avec les ordinateurs et la manière dont les gens enregistrent et maintiennent l'information.

Conserver la trace des choses

Aujourd'hui les gens utilisent les ordinateurs pour accomplir des tâches qui nécessitaient autrefois d'autres outils. Les ordinateurs ont remplacé les machines à écrire pour rédiger et modifier des documents. Ils ont surpassé les calculateurs électromécaniques dans le calcul mathématique. Ils ont aussi remplacé des millions de feuilles de papier et de fichiers dans le stockage d'informations sensibles. Les ordinateurs travaillent mieux et plus rapidement que tous ces vieux outils. Cependant, tout cela a un coût. Les utilisateurs n'ont plus un accès physique direct à leurs données.

Quand les ordinateurs tombent en panne (c'est rare, mais cela peut arriver), il est classique de les remettre en question en se demandant s'ils améliorent réellement quelque chose. C'est qu'autrefois quand un dossier « crashait » par terre, il suffisait de ramasser les papiers et de les ranger dans le dossier. Et hormis les tremblements de terre et autres désastres majeurs, rien ne semblait vraiment pouvoir faire « tomber en panne » une armoire à dossiers. De même, on n'a jamais vu une armoire envoyer un message d'erreur. Mais quand un disque dur « plante », c'est une autre histoire. Vous ne pouvez pas « ramasser » les bits et les octets. Des pannes mécaniques, électriques ou des défaillances humaines font parfois disparaître de manière définitive des données.

Il convient donc de prendre quelques mesures pour vous protéger contre la perte accidentelle de données. Voici quatre critères auxquels doit répondre le système de stockage de l'information que vous choisirez :

- Il doit être rapide et facile de stocker des données, car vous le ferez souvent.
- Le support de stockage doit être fiable. Vous ne voulez pas vous apercevoir un beau matin que vos données ont

disparu comme par magie.

- Il doit être simple et rapide de récupérer des données, quelle que soit la quantité d'informations dont vous disposez.
- Vous devez pouvoir rapidement et facilement récupérer l'information que vous recherchez parmi des tonnes de données.

Les bases de données dignes de ce nom satisfont ces critères. Et si vous souhaitez stocker plus d'une dizaine de données, vous désirerez probablement en utiliser une.

Qu'est-ce qu'une base de données ?

Le terme base de données est tombé dans le langage commun et a perdu son sens initial. Pour certaines personnes, une base de données est un ensemble d'objets hétéroclites (agenda, liste de courses, parchemins magiques…). D'autres personnes lui accordent une signification plus précise.

Dans ce livre, je définirai une base de données comme un ensemble d'enregistrements intégrés, capable de fournir une description de lui-même. Et tout cela implique évidemment le passage par une technologie informatique associée à certains langages (SQL en étant l'exemple qui nous intéresse dans ce livre).

Un enregistrement est la représentation d'une donnée physique ou d'un objet conceptuel. Imaginons par exemple que vous souhaitiez conserver la trace des clients d'une entreprise. Vous assignez un enregistrement à chaque client. Chaque enregistrement dispose de multiples attributs tels que le nom, l'adresse et le numéro de téléphone. Le nom, l'adresse et ainsi de suite constituent les données.

Une base de données est constituée de données et de métadonnées. Une métadonnée est elle-même une donnée qui décrit la structure des données « utiles » présentes dans la base. Si vous savez comment vos données sont arrangées, alors vous pouvez les retrouver. Puisque la base de données contient une description de sa propre structure, on dit qu'elle est autodescriptive. La base de données est intégrée, car elle contient non

seulement des données, mais aussi la description des relations qui les unissent.

La base de données conserve les métadonnées dans une zone nommée dictionnaire des données qui décrit les tables, les colonnes, les index, les contraintes et toute autre caractéristique la définissant.

Comme un système de fichiers plein texte (décrit plus loin dans ce chapitre) ne dispose d'aucune métadonnée, les applications conçues pour travailler avec de tels fichiers doivent contenir dans leur programme des informations équivalant aux métadonnées.

Volume et complexité d'une base de données

Il existe des bases de données de tout volume, d'un simple ensemble de quelques enregistrements au mammouth qui contient des millions d'enregistrements.

- **Une base de données personnelle** est conçue pour être utilisée par une seule personne sur un unique ordinateur. Une telle base adopte généralement une structure relativement simple et son volume est limité.
- **Une base de données départementale** ou de groupe de travail est utilisée par les membres d'un département ou d'un groupe de travail d'une organisation. Ce type de base de données occupe généralement un volume plus important qu'une base de données personnelle et est nécessairement plus complexe, car il doit gérer les accès simultanés de multiples utilisateurs aux mêmes données.
- **Une base de données organisationnelle** peut être énorme. Ce type de base est utilisé pour stocker les informations d'importantes organisations ou entreprises.

Qu’est-ce qu’un système de gestion de bases de données ?

Bonne question. Un système de gestion de bases de données (SGBD) est un ensemble de programmes utilisés pour définir, administrer et traiter des bases de données et leurs applications associées. La base de données « administrée » est, par essence, une structure que vous créez pour contenir des données. Un SGBD est l’outil que vous utilisez pour créer cette structure et traiter les données qui se trouvent dans la base de données.

De nombreux SGBD sont disponibles sur le marché. Quelques-uns ne fonctionnent que sur des grands systèmes (ou mainframes), des ordinateurs portables, des tablettes. Toutefois, la tendance de ces produits est de fonctionner sur de multiples plates-formes ou sur des réseaux qui contiennent différentes classes de machines. Une tendance encore plus marquée est de stocker les données dans des centres d’hébergement de données (*data centers*), voire de les stocker dans le *cloud*, qui peut être un cloud public géré par une entreprise telle qu’Amazon, Google ou Microsoft, via Internet, ou un cloud privé géré par l’organisation qui stocke les données sur son intranet.

De nos jours, *cloud* est un terme à la mode utilisé sans arrêt dans les cercles techniques. Tout comme ces choses blanches dans le ciel, ses contours sont indéfinis et il semble flotter quelque part. En réalité, c’est un ensemble de ressources informatiques accessibles depuis un navigateur, sur Internet ou sur un intranet privé. Ce qui distingue les ressources informatiques dans le cloud de celles d’un data center physique, c’est le fait que ces ressources sont accessibles via un navigateur plutôt que via un programme qui y accède directement.

Un SGBD qui fonctionne sur diverses plates-formes, aussi bien petites que grandes, est dit pouvoir monter en charge.

Quelle que soit la taille de l’ordinateur sur lequel fonctionne la base de données, et que la machine soit connectée ou non à un réseau, le flot des informations entre l’utilisateur et la base de données est toujours le même. La Figure 1.1 montre comment l’utilisateur communique avec la base de données via le SGBD. Le SGBD masque physiquement les détails du stockage des données, de sorte que l’application n’a à se préoccuper que de

la manipulation de celles-ci et non de la manière dont elles sont extraites ou stockées.

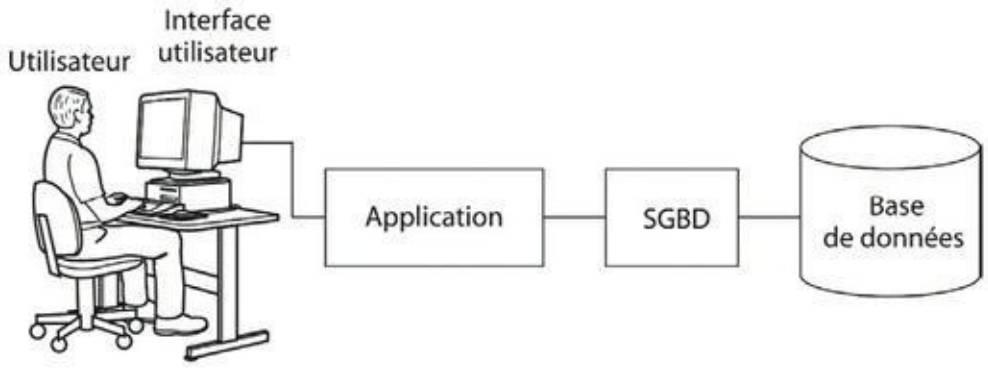


FIGURE 1.1 Diagramme de bloc d'un système d'information à base de SGBD.

LA VALEUR EST DANS LA STRUCTURE, PAS DANS LES DONNÉES

Imaginons un instant que l'on réduise un être humain aux atomes qui le composent (carbone, hydrogène, oxygène, nitrogène, plus quelques autres dont le nom m'échappe). Que resterait-il de vous ? Bien peu de choses en vérité. Mais c'est une mauvaise manière de prendre le problème. Les gens ne sont pas simplement composés de collections d'atomes isolés. Nos atomes se combinent pour former des protéines, des enzymes, des hormones et bien d'autres substances qui coûteraient des millions d'euros le sachet dans l'industrie pharmaceutique. La structure précise de ces combinaisons d'atomes est justement ce qui leur donne une si haute valeur. Par analogie, la structure de nos bases est ce qui rend possible l'interprétation de données n'ayant par elles-mêmes aucune signification précise. La structure ajoute à ces données inertes du sens et de la cohérence. Elle permet d'en dégager des schémas, des trames, des tendances, et ainsi de suite. Des données non structurées (à l'instar d'atomes non combinés) n'ont que peu ou pas de valeur du tout.

Les fichiers plein texte

Quand il s'agit de stocker des données non structurées, le fichier plein texte peut se révéler être une excellente solution. Un tel fichier contient une séquence d'enregistrements présentée dans un unique format et rien d'autre (les données, toutes les données, mais rien que les données). Puisque ce fichier ne contient aucune information sur la structure de l'information (métadonnées), il occupe un espace minimum.

Supposons que vous désiriez stocker les noms et les adresses des clients américains de votre entreprise dans un fichier plein texte. Vous pourriez adopter la structure suivante :

```
Harold Percival   26262 S. Howards Mill Rd
Westminster   CA92683
Jerry Appel       32323 S. River Lane Rd
Santa Anna    CA92705
Adrian Hansen     232 Glenwood Court
Anaheim       CA92640
John Baker        2222 Lafayette St
Garden Grove  CA92643
Michael Pens      77730 S. New Era Rd
Irvine        CA92715
Bob Michimoto     25252 S. Kelmsley Dr
Stanton       CA92610
Linda Smith       444 S.E. Seventh St
Costa Mesa    CA92635
Robert Funnell    2424 Sheri Court
Anaheim       CA92640
Bill Checkal      9595 Curry Dr
Stanton       CA92610
Jed Style         3535 Randall St
Santa Anna    CA92705
```

Comme vous pouvez le voir, le fichier ne contient que les données. Chaque champ occupe un espace fixe (par exemple, le champ Nom

occupe 15 caractères) et aucune structure ne sépare un champ d'un autre. La personne qui a créé cette base de données a décidé de la position et de la longueur des champs. Tout programme qui voudrait utiliser ce fichier devra « connaître » la manière dont les champs sont organisés, puisque le fichier ne contient aucune information à ce sujet.

Du fait de leur simplicité, les fichiers plein texte peuvent offrir une grande rapidité de traitement. Par contre, toute la complexité de la gestion est transférée aux programmes qui les gèrent. Les applications doivent savoir exactement où et comment les données sont enregistrées dans le fichier. C'est pourquoi l'utilisation d'un fichier plein texte n'est intéressante que sur de petits systèmes. Dès que l'on dépasse une certaine échelle, ils deviennent un frein au développement.

Une véritable base de données, si elle contient effectivement un certain nombre d'informations qui ne sont pas directement utiles, présente l'avantage de pouvoir être exploitée sur de nombreuses plates-formes matérielles et systèmes d'exploitation. Il est aussi plus simple d'écrire un programme qui manipule une base de données qu'un fichier plein texte, car le programme n'a pas à connaître à l'avance l'organisation des données qu'il traite.

De plus, la logique d'accès aux données et de leur manipulation n'a pas à être intégrée dans le programme, car elle se trouve déjà dans le SGBD. Par conséquent, il n'est pas nécessaire de reprogrammer cette portion du programme quand il s'agit de porter ce dernier d'une plate-forme matérielle à une autre. Et c'est là où les fichiers plein texte atteignent très vite leurs limites.

Les modèles de base de données

Dans les années 1950, les premières bases de données étaient structurées selon un modèle hiérarchique. Elles posaient des problèmes de redondance de données, et la rigidité de leur structure compliquait leur modification. Bientôt, elles furent suivies par des bases de données qui collaient au modèle réseau, pour tenter d'éliminer les inconvénients principaux du modèle hiérarchique. Les bases de données en réseau minimisent la redondance, mais au prix d'une complexité structurelle accrue.

Quelques années plus tard, le DR. E.F. Codd d'IBM « inventa » le modèle relationnel, qui minimisait la redondance tout en permettant de comprendre

facilement la structure. Le langage SQL a été conçu pour travailler sur des bases de données relationnelles. L'apparition de telles bases permit de faire passer les bases de données hiérarchiques et réseau aux oubliettes de l'histoire.

Un nouveau phénomène est l'émergence de bases de données désignées comme NoSQL, qui n'ont pas la structure des bases de données relationnelles et qui n'utilisent pas le langage SQL. Je ne traite pas des bases de données NoSQL dans ce livre.

Le modèle relationnel

Le Dr E.F. Codd d'IBM « inventa » le modèle relationnel de base de données en 1970, mais ce modèle ne commença à être utilisé dans des produits qu'une dizaine d'années plus tard. Ironie de l'histoire, ce ne fut pas à IBM que revint la primeur de délivrer le premier de ces produits. L'acte de naissance fut signé par une toute jeune société qui nomma son bébé Oracle.

Les bases de données relationnelles ont remplacé les anciens schémas, car le modèle relationnel présente un certain nombre d'avantages sur ceux qui l'ont précédé. Le plus important de ces avantages est sans doute que, dans une base de données relationnelle, vous pouvez changer la structure de cette base sans pour autant avoir à modifier les applications basées sur les anciennes structures. Supposons par exemple que vous ajoutiez une ou plusieurs nouvelles colonnes à une table de base de données. Vous n'aurez pas besoin de changer les applications précédemment écrites pour traiter cette table, à moins que vous n'ayez modifié une ou plusieurs des colonnes manipulées par lesdites applications.

Bien entendu, si vous supprimez une colonne référencée par une application existante, cette dernière risque de ne plus fonctionner correctement. Et c'est vrai quel que soit le modèle de base de données que vous utilisez. Il n'y a pas de meilleure manière pour « planter » une application de base de données que de lui demander de récupérer une information qui n'existe plus.

Les composants d'une base de données relationnelle

Les bases de données relationnelles tirent leur flexibilité du fait que leurs données résident dans des tables largement indépendantes les unes des autres. Vous pouvez ajouter, supprimer ou modifier des données dans une table sans modifier celles qui se trouvent dans les autres tables, du moment que la structure affectée n'est pas un parent d'autres tables (les relations parent-enfant entre les tables sont expliquées dans le Chapitre 5). Dans cette section, je vais vous montrer en quoi consistent ces tables et comment elles sont liées aux autres éléments qui composent une base de données relationnelle.

Devine qui vient dîner ce soir ?

Nombreuses sont les relations que j'invite à ma table durant les vacances. Les bases de données ont aussi des relations, mais chacune de leur relation dispose de sa propre table. Une base de données relationnelle est constituée d'une ou de plusieurs relations.

Une *relation* est un tableau à *deux dimensions*, donc formé de lignes et de colonnes. Chaque cellule du tableau contient une et une seule valeur, et deux lignes ne peuvent pas être identiques.

Les tableaux à deux dimensions sont ceux que vous utilisez dans des tableurs de type Microsoft Excel ou OpenOffice.org Calc. Les statistiques d'un joueur de base-ball que l'on retrouve au dos de cartes de collection sont un autre exemple de tels tableaux. On trouve sur ces cartes des colonnes pour l'année, l'équipe, les parties, les différents types de points marqués. Une rangée traite d'une année durant laquelle le joueur aura participé aux événements sportifs majeurs. Vous pouvez stocker ces données dans une relation (une table) qui adopterait la même structure de base. La Figure 1.2 représente une table d'une base de données relationnelle qui contient les statistiques d'un certain joueur. Dans la pratique, ce genre de table proposerait des statistiques relatives à tous les joueurs d'une équipe.

Joueur	Année	Equipe	Partie	Moy. Bat.	Hits	Runs	RBI	2B	3B	HR	Walk	Steals	Moy. Bat.
Roberts	1988	Padres	5	9	3	1	0	0	0	0	1	0	.333
Roberts	1989	Padres	117	329	99	81	25	15	8	3	49	21	.301
Roberts	1990	Padres	149	556	172	104	44	36	3	9	55	46	.309

FIGURE 1.2 Une table contenant les statistiques d'un joueur de base-ball.

Les colonnes d'un tableau sont *consistantes*, c'est-à-dire qu'une colonne a la même signification dans chaque ligne. Si une colonne contient le nom d'un joueur dans une ligne, elle devra contenir le nom d'un joueur dans toutes les lignes. L'ordre dans lequel les lignes et les colonnes apparaissent dans le tableau n'a aucune signification. Le SGBD traite la table de la même manière, quel que soit l'ordre de ses colonnes. Il en va de même pour les lignes.

Chaque colonne de la table d'une base de données représente un attribut de la table. La signification d'une colonne est la même pour toutes les lignes de la table. Par exemple, une table peut contenir les noms, adresses et numéros de téléphone de tous les clients d'une organisation. Chaque ligne de la table (aussi nommée *enregistrement* ou *tuple*) contient les informations relatives à un seul client. Chaque colonne représente un unique attribut, tel que le numéro du client, le nom du client, la rue du client, la ville du client, le code postal du client ou encore son numéro de téléphone. La Figure 1.3 montre quelques-unes des lignes et des colonnes d'une telle table.

FIGURE 1.3 Chaque ligne de la base de données contient un enregistrement ; chaque colonne de la base de données contient un unique attribut.

Dans ce modèle de base de données, les *relations* correspondent aux *tables* figurant dans toute base de données fondée sur ce modèle. Répétez-le dix fois rapidement pour vous en souvenir.

Profitez de la vue

Il vous est probablement déjà arrivé de vous arrêter sur la route pour contempler la vue. Les bases de données ont aussi des vues, même si elles ne sont pas aussi pittoresques. Les vues des bases de données tirent leur beauté du fait qu'elles sont extrêmement utiles quand vous travaillez sur vos données.

Les tables peuvent contenir de nombreuses colonnes et lignes. Il arrive que toutes les données vous intéressent, et il arrive aussi que ce ne soit pas le cas. Seules quelques colonnes de la table retiennent votre attention ou seules quelques lignes répondent à vos conditions. Certaines colonnes d'une table et certaines colonnes d'une autre table peuvent vous intéresser. Pour éliminer les données dont vous n'avez pas besoin dans l'instant, vous pouvez créer une *vue*. Une vue est un sous-ensemble d'une base de données qu'une application est capable de traiter. Elle peut contenir des parties d'une ou de plusieurs tables.

Les vues sont parfois appelées *tables virtuelles*. Du point de vue de l'application ou de l'utilisateur, les vues se comportent comme des tables. Pour autant, elles n'ont pas d'existence physique. Les vues vous permettent de jeter un œil sur vos données, mais elles ne font pas partie des données en elles-mêmes.

Imaginons par exemple que vous travailliez sur une base de données qui contient une table CLIENTS et une table FACTURES. La table CLIENTS possède les colonnes `CLIENT_ID`, `NOM`, `PRENOM`, `RUE`, `VILLE`, `ETAT`, `CODE_POSTAL` et `TELEPHONE`. La table FACTURES contient les colonnes `NUMERO_FACTURE`, `CLIENT_ID`, `DATE`, `TOTAL_VENTE`, `TOTAL_REMIS` et `METHODE_PAIEMENT`.

Un responsable des ventes du pays veut voir à l'écran le prénom, le nom et le numéro de téléphone des clients. Vous pouvez créer à partir de la table CLIENTS une vue qui contienne seulement ces trois colonnes et permette au responsable de n'afficher que les informations dont il a besoin, sans s'encombrer avec le contenu des autres colonnes. La Figure 1.4 montre à quoi ressemblerait cette vue.

Un responsable en charge d'un département ou d'un état pourrait vouloir ne visualiser que les noms et les numéros de téléphone des clients dont le code postal est compris entre 90000 et 93999 (Californie centrale et Californie du Sud). Une vue qui appliquerait une restriction sur les lignes et les colonnes voulues pourrait effectuer ce travail. La Figure 1.5 montre d'où cette vue tirerait ses données.

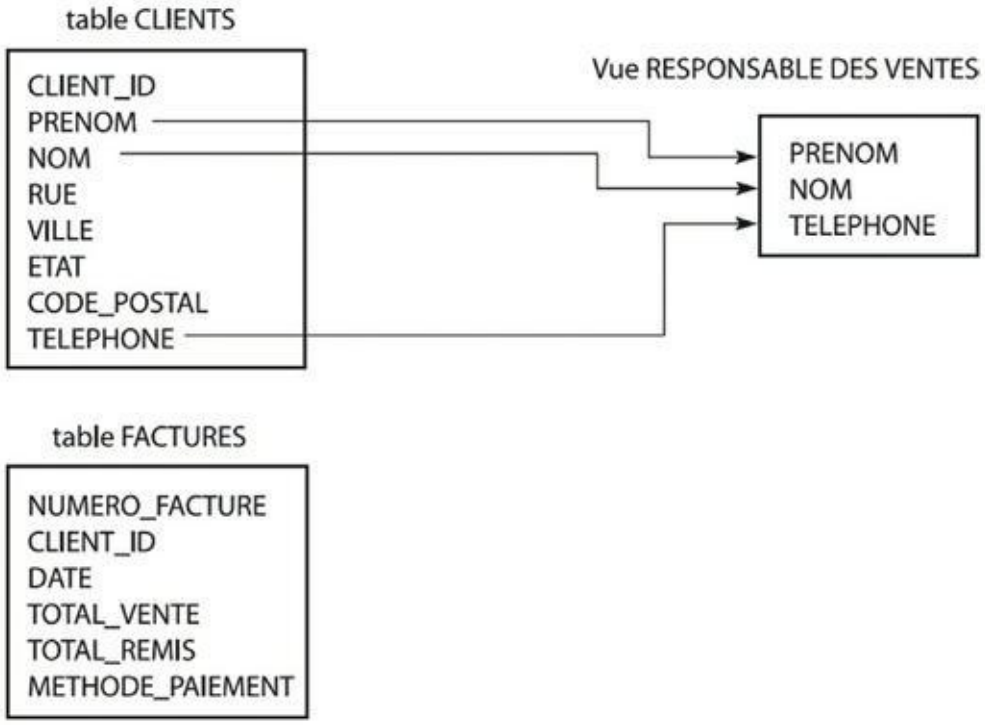


FIGURE 1.4 La vue du RESPONSABLE DES VENTES dérive de la table CLIENTS.

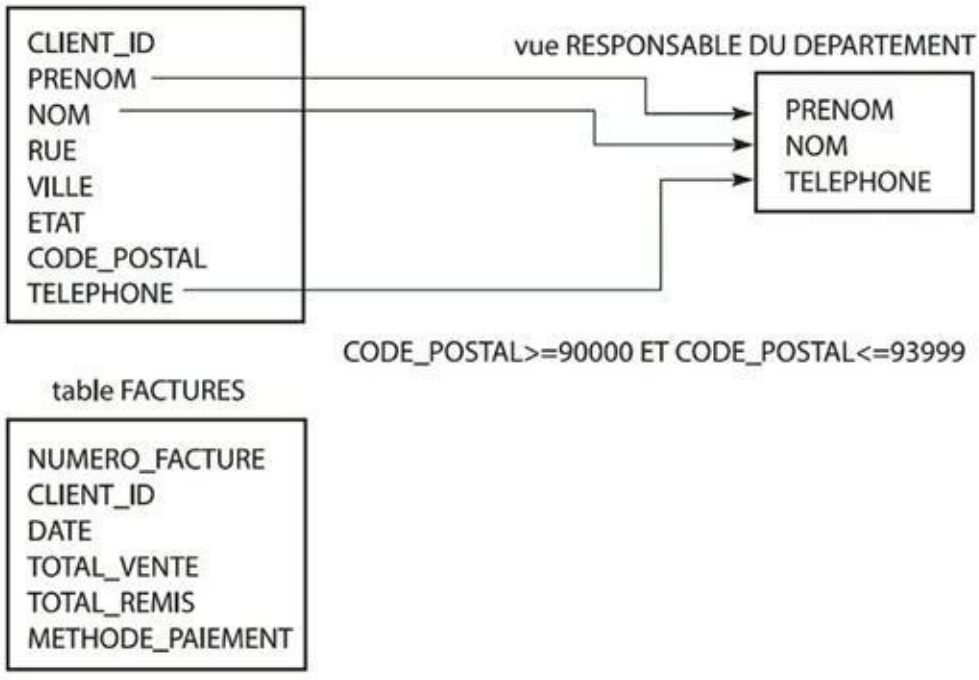


FIGURE 1.5 La vue du RESPONSABLE DU DEPARTEMENT ne contient que quelques-unes des lignes de la table CLIENTS.

Le responsable des ventes en cours peut vouloir visualiser les données `NOM` et `PRENOM` de la table CLIENTS ainsi que les données `DATE`,

`TOTAL_VENTE`, `TOTAL_REMIS` et `METHODE_PAIEMENT` de la table FACTURES, quand `TOTAL_REMIS` vaut moins que `TOTAL_ VENTE`, ce qui sera le cas si le client n'a pas encore entièrement réglé sa facture. Il faut donc élaborer une vue qui contienne des données extraites des deux tables. La Figure 1.6 montre le flux de données entre la vue du responsable des ventes en cours et les tables CLIENTS et FACTURE.

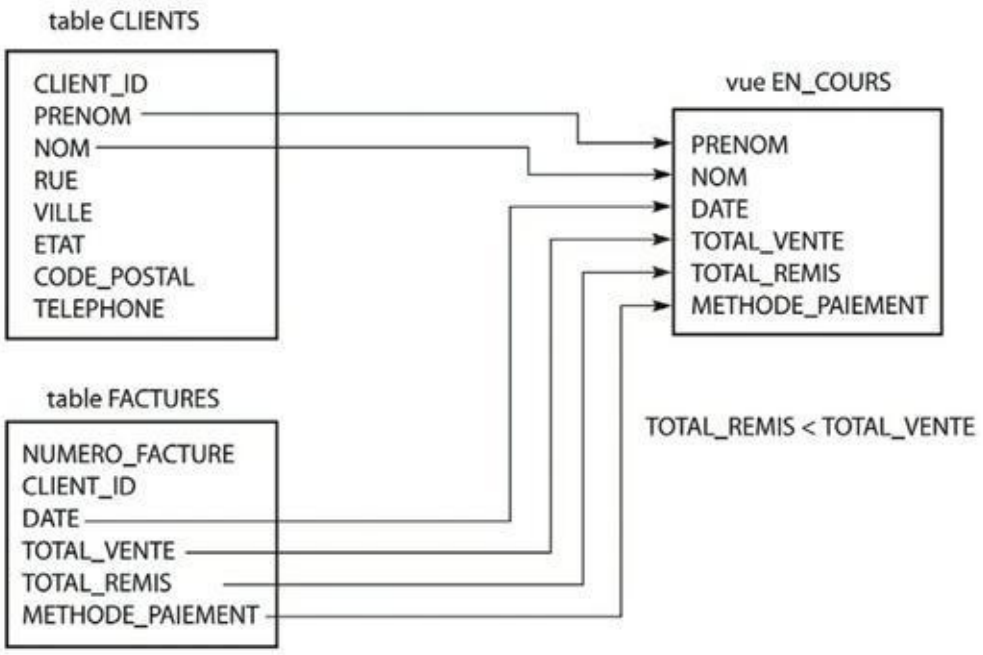


FIGURE 1.6 La vue EN_ COURS tire ses données de deux tables.

Les vues sont très utiles car elles vous permettent d'extraire et de formater des données d'une base de données sans altérer physiquement les données stockées. Elles protègent aussi les données que vous *ne voulez* pas afficher, car elles ne les contiennent pas. Le Chapitre 6 explique dans le détail comment créer une vue en SQL.

Schémas, domaines et contraintes

Une base de données n'est pas seulement un ensemble de tables. De nombreuses structures supplémentaires, sur différents niveaux, permettent de maintenir l'intégrité des données. Le *schéma* de la base de données détaille l'organisation générale des tables. Le *domaine* d'une colonne d'une table vous indique quelles valeurs vous pouvez stocker dans cette colonne. Vous pouvez appliquer des *contraintes* à une table d'une base de données

pour empêcher que quelqu'un (dont vous-même) n'enregistre des données non valides dans la table.

Schémas

L'intégralité de la structure d'une base de données est stockée dans son *schéma* ou *vue conceptuelle*. Cette structure est quelquefois aussi appelée *vue logique* complète de la base de données. Le schéma est une métadonnée et, en tant que telle, une partie intégrante de la base de données. La métadonnée, qui décrit la structure de la base de données, est stockée dans des tables qui ressemblent en tout point aux tables qui contiennent des données classiques. Une métadonnée n'est rien de plus qu'une donnée. C'est la beauté de la chose.

Domaines

Un attribut d'une relation (autrement dit une colonne d'une table) peut n'accepter qu'un nombre fini de valeurs. L'ensemble de ces valeurs autorisées constitue le *domaine* de l'attribut.

Supposons que vous soyez un vendeur de voitures et que vous deviez gérer le nouveau modèle Curarri GT 4000 coupé sport. Vous conservez la trace des voitures que vous avez en stock dans une table que vous nommez INVENTAIRE. Vous nommez l'une des colonnes de cette table COULEUR, car elle contient la couleur extérieure de chaque voiture. La GT 4000 existe en quatre coloris : gris, noir, blanc et rouge. Ces quatre couleurs constituent le domaine de l'attribut COULEUR.

Contraintes

Une *contrainte* est un élément important (quoique trop souvent négligé) d'une base de données. C'est une règle qui précise les valeurs que peut prendre un attribut d'une table.

En appliquant des contraintes à une colonne, vous pouvez empêcher les gens de saisir des données non valides dans cette colonne. Bien entendu, chacune des valeurs appartenant au domaine de la colonne doit satisfaire les contraintes qui pèsent sur cette dernière. Comme je l'ai mentionné dans la section précédente, le domaine d'une colonne est l'ensemble des valeurs que cette colonne peut contenir. Une contrainte est une restriction sur ce que

la colonne a le droit de contenir. Les caractéristiques d'une colonne de table, plus les contraintes qui s'appliquent à cette colonne, déterminent le domaine de la colonne. En appliquant des contraintes, vous pouvez empêcher la saisie d'une donnée qui n'appartient pas au domaine de la colonne.

Dans l'exemple du vendeur de voitures, vous pouvez contraindre la base de données à n'accepter que les quatre couleurs du modèle dans la colonne `COULEUR`. Si l'opérateur de saisie tente de rentrer une autre teinte, par exemple vert, le système refusera la saisie.

Vous vous demandez déjà ce qui va se passer si le constructeur décide de sortir une série limitée de la Curarri GT 4000 coupé sport verte ! La réponse est simple (roulez tambours) : il va falloir que les développeurs chargés de la maintenance de la base de données modifient le domaine de la colonne `COULEUR` pour s'adapter au nouveau modèle. Ce genre de situation se produit tout le temps, impliquant donc des adaptations de la structure de la base de données. Seules les personnes qui savent comment procéder (et vous en faites partie) seront capables d'éviter un désastre commercial.

Le modèle objet opposé au modèle relationnel

Le modèle relationnel s'est révélé extraordinairement utile dans un grand nombre d'applications. Cependant, il n'est pas exempt de défauts. Ces problèmes sont devenus plus évidents avec l'avènement des langages de programmation orientés objet tels que le C++, Java et C#. De tels langages sont capables de s'attaquer à des questions bien plus complexes que les langages traditionnels, en particulier parce qu'ils implémentent des concepts tels que l'extensibilité des types de données, l'encapsulation, l'héritage, le lien dynamique aux méthodes, les objets complexes et composites et l'identité des objets.

Je ne vais pas expliquer dans ce livre à quoi correspondent toutes ces notions (même si nous ferons référence à certaines d'entre elles ultérieurement). Il faut simplement comprendre que le modèle relationnel classique n'est pas compatible avec un bon nombre de ces fonctionnalités. Il en résulte que des systèmes de gestion de bases de données basés sur le

modèle objet ont été développés et sont aujourd'hui commercialisés. Cependant, leur part de marché est encore limitée à ce jour.

Score final : bases de données relationnelles : 1 ; bases de données orientées objet : 0.

Le modèle objet-relationnel

Les concepteurs de bases de données sont constamment en quête du meilleur monde possible. Ils songent : « Ne serait-il pas fabuleux si nous pouvions disposer de tous les avantages que procurent les systèmes de gestion de bases de données orientés objet tout en restant compatibles avec le système relationnel que nous connaissons et apprécions tous ? » Ce type de réflexion a mené au modèle objet-relationnel. Les SGBD objet-relationnel étendent le modèle relationnel en y intégrant la gestion de la modélisation orientée objet des données. Des fonctionnalités orientées objet ont été rajoutées au standard international SQL pour permettre aux vendeurs de SGBD relationnels de transformer leurs produits en SGBD orientés objet, tout en restant compatibles avec le standard. Si le standard SQL-92 ne traite que du modèle purement relationnel, le standard SQL-1999 (également appelé SQL3) décrit un modèle de bases de données objet-relationnel. SQL:2003 inclut encore plus de fonctionnalités orientées objet, et les versions suivantes du standard SQL ont poussé encore plus loin dans ce sens.

Je présente dans ce livre le standard international SQL ISO/ IEC. C'est-à-dire, et pour l'essentiel, un modèle de bases de données relationnelles. Je décris aussi les extensions orientées objet qui ont été rajoutées dans SQL:1999 ainsi que celles introduites dans les versions suivantes. Les fonctionnalités orientées objet du nouveau standard permettent aux développeurs d'utiliser des bases de données SQL pour résoudre des problèmes trop complexes à résoudre pour les anciens concepts purement relationnels. Les vendeurs de SGBD incorporent les fonctionnalités orientées objet du standard ISO dans leurs produits. Quelques-unes de ces fonctionnalités existent depuis des années, mais d'autres restent encore à inclure.

Considérations sur la conception de bases de données

Une base de données est la représentation d'une structure physique ou conceptuelle telle qu'une organisation, une usine de voitures ou les statistiques des performances de tous les clubs de base-ball. La précision de la représentation dépend du niveau de détail apporté lors de la conception. La somme d'efforts que vous investissez dans la conception de votre base de données devrait dépendre du type d'information que vous comptez y stocker. Trop de détails rime avec efforts inutiles, temps perdu et gaspillage d'espace sur le disque. Pas assez de détails peut rendre votre base de données inutilisable.

Etudiez le niveau de détail dont vous avez besoin sur l'instant et celui dont vous aurez besoin dans le futur. C'est à l'avenir que vous devez penser lors de votre conception. Mais ne soyez pas surpris si vous devez procéder à quelques ajustements pour satisfaire aux contraintes du monde réel et à leurs évolutions.

Les systèmes de gestion de bases de données actuels, dotés d'interfaces graphiques attrayantes et d'outils de conception intuitifs, peuvent entretenir chez le concepteur un faux sentiment de sécurité. Ces systèmes laissent à penser que la conception d'une base de données s'apparente beaucoup à celle d'une feuille de calcul dans un tableur. Malheureusement, ce n'est pas le cas. La conception d'une base de données est une tâche complexe. Si vous vous y prenez mal, vous produirez une base de données qui ne résistera pas au temps. Bien souvent, le problème n'apparaît que lorsque vous avez dépensé beaucoup d'énergie à saisir les données. Et quand vous réalisez enfin que ce problème existe, il a généralement pris une telle ampleur que la seule solution est de reprendre à zéro la conception de la base de données et la saisie de ces dernières. Le bon côté des choses est que cela vous sert de leçon en vous poussant à construire une seconde version de bien meilleure qualité que la première

Chapitre 2

Les bases de SQL

DANS CE CHAPITRE :

- **Comprendre SQL.**
- **Se défaire de quelques idées préconçues sur SQL.**
- **Jeter un œil sur les différents standards SQL.**
- **Se familiariser avec les commandes communes de SQL et les mots réservés.**
- **Représenter des nombres, des caractères, des dates, des heures et d'autres types de données.**
- **Etudier les valeurs nulles et les contraintes.**
- **Mettre SQL en marche sur un système client/serveur.**
- **SQL en réseau.**

SQL est un langage souple que vous pouvez utiliser de différentes manières. C'est l'outil le plus employé pour communiquer avec une base de données relationnelle. Dans ce chapitre, j'expliquerai ce qu'est SQL et ce qu'il n'est pas, en le distinguant plus particulièrement des autres langages. Puis, je décrirai les commandes et les types de données que SQL supporte dans sa version standard avant de dévoiler quelques concepts clés : les valeurs nulles et les contraintes. Enfin, je présenterai rapidement la place que SQL occupe dans un environnement client/serveur, ainsi que sur Internet et sur des intranets.

Ce que SQL n'est pas

La première chose qu'il faut comprendre quand on parle de SQL est que ce n'est pas un langage procédural comme le Basic, le C, le C++, le C# et Java. Pour résoudre un problème à l'aide d'un de ces langages procéduraux,

vous écrivez une procédure qui effectue une opération spécifique, puis une autre, et ce jusqu'à ce que la tâche soit accomplie. La procédure peut être une séquence linéaire ou boucler sur elle-même, mais dans les deux cas le programmeur spécifie l'ordre d'exécution.

SQL est *non procédural*. Pour résoudre un problème avec SQL, vous lui indiquez simplement ce que vous voulez (à l'égal d'Aladin et de sa lampe magique) au lieu d'indiquer au système comment récupérer ce que vous voulez. Le système de gestion de bases de données (SGBD) décide de la meilleure manière d'exaucer votre souhait.

Je viens de vous dire que SQL n'est pas un langage procédural. C'est dans l'ensemble vrai. Cependant, des millions de programmeurs sont habitués à résoudre des problèmes d'une manière procédurale. C'est pourquoi il y a eu ces dernières années une pression considérable pour que de telles fonctionnalités soient ajoutées à SQL. C'est pourquoi ce langage comprend aujourd'hui des instructions qui se rattachent à la programmation procédurale telles que les blocs `BEGIN`, l'instruction `IF`, les fonctions et les procédures. Tout cela a été ajouté de manière à pouvoir stocker sur le serveur des programmes utilisables par de multiples utilisateurs.

Pour illustrer ce que je veux dire par « indiquer au système ce que vous voulez », supposez que vous disposiez d'une table EMPLOYES et que vous souhaitiez y récupérer les lignes correspondant à toutes les personnes « senior ». Selon vous, un senior se définit comme quelqu'un dont l'âge dépasse 40 ans ou qui gagne plus de 60 000 € par an. Vous pouvez récupérer les informations que vous recherchez en utilisant la requête suivante :

```
SELECT * FROM EMPLOYES WHERE AGE>40 OR
SALAIRE>600000 ;
```

Cette requête retourne toutes les lignes de la table EMPLOYES dans lesquelles ou bien la valeur de la colonne AGE est supérieure à 40 ou bien la valeur de la colonne SALAIRE est supérieure à 60 000 (ou bien les deux à la fois). En SQL, vous n'avez pas besoin de préciser comment l'information doit être récupérée. Le moteur de la base de données examine la base et décide par lui-même de la manière de répondre à votre requête. Vous n'avez besoin que de spécifier quelles données vous recherchez.

Une requête est une question que vous posez à la base de données. Si une donnée contenue dans la base répond aux conditions que vous formulez dans

votre requête, SQL vous renverra cette donnée.

Les implémentations actuelles de SQL ne comportent pas la plupart des constructions de base présentes dans les autres langages de programmation. Les applications du monde réel utilisent généralement au moins quelques-unes de ces constructions, c'est pourquoi SQL est qualifié de sous-langage de données. Même avec les extensions ajoutées en 1999, 2003, 2005 et 2008, vous devrez utiliser SQL en combinaison avec un langage procédural, tel que le C++, pour produire des applications complètes.

Vous pouvez extraire de l'information à partir d'une base de données de deux manières :

- **En formulant une requête *ad hoc* depuis la console de l'ordinateur en tapant les instructions SQL puis en lisant le résultat à l'écran**. La console est le terme traditionnellement utilisé pour désigner le matériel qui joue le rôle de clavier et d'écran sur les systèmes PC. Vous pouvez formuler ainsi vos requêtes si vous souhaitez obtenir rapidement une réponse à une question spécifique. Vous n'avez jamais eu besoin d'une certaine information et vous n'en aurez très vraisemblablement plus jamais besoin. Mais ce jour-ci, à cette heure-ci, c'est exactement ce qu'il vous faut. Tapez la requête SQL appropriée sur votre clavier, et la réponse va s'afficher sur l'écran en moins de temps qu'il n'en faut pour le dire.
- **En exécutant un programme qui récupère l'information dans la base de données et produit un rapport soit à l'écran, soit sur une imprimante**. Ajouter une requête directement dans votre programme est la meilleure solution si cette requête est complexe et si vous comptez l'utiliser plusieurs fois. Vous n'avez ainsi à la formuler qu'une fois, puis à y faire appel aussi souvent que

> vous le voulez. Le Chapitre 16 vous explique comment incorporer du code SQL dans des programmes écrits dans un autre langage.

Un (tout) petit peu d'histoire

SQL est né dans les laboratoires de recherche d'IBM, tout comme la théorie des bases de données relationnelles. Au début des années 1970, alors que les chercheurs d'IBM travaillaient sur ce qui allait devenir les SGBD relationnels (ou SGBDR), ils créèrent un sous-langage de données pour travailler sur ces systèmes. Ils nommèrent la première version de ce langage *SEQUEL* (Structure English QUEry Language ou Langage de requêtes structuré). Cependant, quand vint le moment de distribuer ce langage sous la forme d'un produit, ils voulurent s'assurer que les gens comprendraient en quoi leur produit était différent de toutes les précédentes versions de SGBD. C'est pourquoi ils décidèrent de donner au nouveau produit un nom qui soit différent de SEQUEL, mais qui soit assimilable à un produit de la même famille. Ils le nommèrent donc SQL.

La syntaxe de SQL est une forme d'anglais structuré, et c'est de là qu'il tient son nom. Toutefois, SQL n'est pas un *langage* structuré au sens où les informaticiens l'entendent. C'est pourquoi, en dépit de ce qui est répandu, SQL n'est pas l'acronyme de « langage de requête structuré ». C'est simplement une séquence de trois lettres sans signification, comme le nom du langage C qui ne signifie rien.

Les travaux d'IBM sur les bases de données relationnelles et sur SQL devinrent très populaires dans l'industrie avant même qu'IBM n'introduise son SGBD SQL/DS en 1981. À cette époque, Relational Software Inc. (aujourd'hui Oracle Corporation) avait déjà distribué son premier SGBD. À peine nés, ces produits imposèrent immédiatement un standard pour une nouvelle classe de systèmes de gestion de bases de données. Ils incorporaient SQL, qui devint de facto le standard des sous-langages de données. Les éditeurs d'autres systèmes de gestion de bases de données produisirent leurs propres versions de SQL. Ces versions comportaient toutes les fonctionnalités des produits d'IBM en y rajoutant des extensions pour tirer le meilleur parti du SGBD sous-jacent. Puisque tous les éditeurs utilisaient une forme particulière de SQL, la compatibilité entre les différents SGBD du marché était évidemment faible.

Une *implémentation* est un SGBDR particulier qui fonctionne sur une plate-forme matérielle spécifique.

Rapidement, un mouvement se créa pour réaliser un standard universellement reconnu de SQL auquel tous les éditeurs pourraient adhérer. En 1986, l'ANSI publia un standard formel nommé SQL-86. L'ANSI mit à jour son standard en 1989 (SQL-89) puis en 1992 (SQL-92). Chaque fois que les éditeurs de SGBDR sortaient de nouvelles versions de leurs produits, ils tentaient de coller toujours plus à ces spécifications. C'est pourquoi, finalement, SQL est devenu un langage véritablement standard et portable.

La dernière version complète du standard SQL est SQL:2011 (ISO/IEC 9075X : 2011). Dans ce livre, je traite de SQL comme le standard SQL:2011 le décrit. Toute implémentation spécifique de SQL s'écarte bien entendu plus ou moins de ce standard. Mais les éditeurs de logiciels tentent de supporter un sous-ensemble essentiel et cohérent du langage SQL standard. La version complète du standard ISO/IEC est proposée à la vente sur `http://webstore.ansi.org`, mais vous ne souhaiterez sans doute pas l'acquérir à moins que vous n'entrepreniez de créer votre propre système de gestion de bases de données fondé sur le standard. Le standard est *hautement* technique, c'est-à-dire virtuellement incompréhensible pour qui n'a pas étudié les langages informatiques.

Les commandes SQL

Le langage SQL comporte un nombre limité de commandes qui sont spécifiquement liées à la manipulation des données. Quelques-unes de ces commandes sont des fonctions de définition de données, d'autres sont des fonctions de manipulation de données, et d'autres encore des fonctions de contrôle de données. Je traite des commandes de définition et de manipulation de données dans les Chapitres 4 à 13 et des commandes de contrôle de données dans les Chapitres 14 et 15.

Pour se conformer au standard SQL:2011, une implémentation doit comporter toutes les fonctionnalités de base. Elle peut aussi inclure certaines des extensions à ce noyau dur (comme le décrit la spécification SQL:2011). Le Tableau 2.1 liste les commandes de base de SQL:2011. Si vous êtes de ces programmeurs qui aiment essayer de nouvelles fonctionnalités, rejoignez l'équipe !

TABLEAU 2.1 Les commandes de SQL:2011.

ADD	DEALLOCATE PREPARE	FREE LOCATOR
ALLOCATE CURSOR	DECLARE	GET DESCRIPTOR
ALLOCATE DESCRIPTOR	DECLARE LOCAL TEMPORARY TABLE	GET DIAGNOSTICS
ALTER DOMAIN	DELETE	GRANT PRIVILEGE
ALTER ROUTINE	DESCRIBE INPUT	GRANT ROLE
ALTER SEQUENCE GENERATOR	DESCRIBE OUTPUT	HOLD LOCATOR
ALTER TABLE	DISCONNECT	INSERT
ALTER TRANSFORM	DROP	MERGE
ALTER TYPE	DROP ASSERTION	OPEN
CALL	DROP ATTRIBUTE	PREPARE
CLOSE	DROP CAST	RELEASE SAVEPOINT
COMMIT	DROP CHARACTER SET	RETURN
CONNECT	DROP COLLATION	REVOKE
CREATE	DROP COLUMN	ROLLBACK
CREATE ASSERTION	DROP CONSTRAINT	SAVEPOINT
CREATE CAST	DROP DEFAULT	SELECT
CREATE CHARACTER SET	DROP DOMAIN	SET CATALOG
CREATE COLLATION	DROP METHOD	SET CONNECTION

CREATE DOMAIN	DROP ORDERING	SET CONSTRAINTS
CREATE FUNCTION	DROP ROLE	SET DESCRIPTOR
CREATE METHOD	DROP ROUTINE	SET NAMES
CREATE ORDERING	DROP SCHEMA	SET PATH
CREATE PROCEDURE	DROP SCOPE	SET ROLE
CREATE ROLE	DROP SEQUENCE	SET SCHEMA
CREATE SCHEMA	DROP TABLE	SET SESSION AUTHORIZATION
CREATE SEQUENCE	DROP TRANSFORM	SET SESSION CHARACTERISTICS
CREATE TABLE	DROP TRANSLATION	SET SESSION COLLATION
CREATE TRANSFORM	DROP TRIGGER	SET TIME ZONE
CREATE TRANSLATION	DROP TYPE	SET TRANSACTION
CREATE TRIGGER	DROP VIEW	SET TRANSFORM GROUP
CREATE TYPE	EXECUTE IMMEDIATE	START TRANSACTION
CREATE VIEW	FETCH	UPDATE
DEALLOCATE DESCRIPTOR		

Les mots réservés

En plus de ces commandes, un certain nombre d'autres mots ont une signification particulière en SQL et sont réservés à des usages spécifiques. Vous ne pouvez donc pas les employer pour nommer des variables ou à toute autre fin. Vous comprendrez facilement pourquoi des tables, des colonnes ou des variables ne peuvent porter des noms qui correspondent à des mots réservés. Imaginez quelle confusion une instruction telle que la suivante pourrait induire :

```
SELECT SELECT FROM SELECT WHERE SELECT = WHERE ;
```

L'Annexe A contient une liste complète des mots réservés de SQL.

Les types de données

Différentes implémentations de SQL gèrent différents types de données. La spécification SQL:2003 reconnaît six types prédéfinis généraux :

- Les numériques.
- Les binaires.
- Les chaînes.
- Les booléens.
- Les dates/heures.
- Les intervalles.
- XML.

Chacun de ces types peut contenir plusieurs sous-types (les numériques exacts, les numériques approximatifs, les chaînes de caractères, les chaînes de bits, les chaînes d'objets). En plus de ces types prédéfinis, SQL gère les collections, les constructions et les types définis par l'utilisateur. Nous y reviendrons plus loin dans ce chapitre.

Si vous utilisez une implémentation de SQL qui gère un ou plusieurs types de données que la spécification SQL : ne mentionne pas, vous pouvez rendre votre base de données plus portable en évitant d'y recourir. Avant que vous ne décidiez de créer et d'utiliser des types de données définis par

l'utilisateur, assurez-vous que tous les SGBD que vous pourriez vouloir gérer à l'avenir supportent aussi ces types.

Les numériques exacts

Comme vous l'aurez probablement deviné d'après leur nom, les types de données numériques exacts vous permettent d'exprimer exactement la valeur d'un nombre. Cinq types de données tombent dans cette catégorie :

- INTEGER
- SMALLINT
- BIGINT
- NUMERIC
- DECIMAL

Le type de données INTEGER

Une donnée de type INTEGER n'a pas de partie décimale et sa précision dépend de chaque implémentation spécifique de SQL. En tant que développeur, vous ne pouvez donc pas spécifier la précision que vous souhaitez.

La *précision* d'un nombre est la quantité de bits sur laquelle ce nombre est codé.

Le type de données SMALLINT

Le type de données `SMALLINT` sert aussi pour les entiers, mais la précision accordée à `SMALLINT` sur une implémentation spécifique ne peut dépasser celle de `INTEGER` sur cette même version. Les implémentations pour systèmes IBM/370 représentent généralement les `SMALLINT` et les `INTEGER` respectivement à l'aide de 16 et de 32 bits. Bien souvent, les précisions de `SMALLINT` et de `INTEGER` sont les mêmes.

Si vous définissez une colonne d'une table de base de données comme étant une donnée entière et que vous savez que la plage des valeurs de la colonne ne dépassera jamais la précision du type de données `SMALLINT` pour l'implémentation que vous utilisez, vous pouvez alors utiliser ce type plutôt que `INTEGER`. Cela permettra à votre SGBD d'économiser de la place.

Le type de données BIGINT

`BIGINT` se définit comme étant un type dont la précision est au moins aussi grande que `INTEGER`. Bien entendu, cette précision dépend de l'implémentation.

Le type de données NUMERIC

Le type de données `NUMERIC` permet de représenter des nombres qui comportent une partie décimale. Vous pouvez spécifier et la précision et l'échelle de la donnée `NUMERIC` (rappelez-vous que la précision est le nombre maximal de bits possibles).

L'échelle d'un nombre est la quantité de chiffres que comporte sa partie décimale. L'échelle d'un nombre ne peut être ni négative ni plus grande que la précision de ce nombre.

Si vous spécifiez le type de données `NUMERIC`, votre implémentation de SQL vous donnera exactement le nombre de bits et l'échelle que vous souhaitez. Vous pourriez spécifier simplement `NUMERIC` et utiliser l'échelle et la précision par défaut, ou `NUMERIC (p)` et définir alors la précision que vous demandez et l'échelle par défaut, ou encore `NUMERIC (p, e)` et spécifier la précision et l'échelle que vous souhaitez. Les paramètres p et e sont à remplacer par les valeurs que vous souhaitez utiliser.

Supposons par exemple que la précision par défaut du type de données `NUMERIC` de votre implémentation de SQL soit 12 et que l'échelle par défaut soit 6. Si vous assignez à une colonne de la base de données le type de données `NUMERIC`, la colonne pourra contenir des nombres dont la valeur n'excédera pas 999 999,999999. Si vous spécifiez le type de données `NUMERIC (10)` pour cette colonne, la colonne pourra contenir

des nombres dont la valeur ne dépassera pas 9 999,999999. Le paramètre `(10)` spécifie la quantité maximale de chiffres possible dans le nombre. Si vous spécifiez le type de données `NUMERIC (10,2),` cette colonne pourra contenir des nombres dont la valeur n'excédera pas 99 999 999,99. Dans ce cas vous disposez toujours de 10 chiffres mais deux seulement sont dédiés à la partie décimale.

Les données `NUMERIC` sont à utiliser pour stocker des nombres tels que 595,72. Cette valeur a une précision de 5 (le nombre total de chiffres) et une échelle de 2 (la quantité de chiffres qui se trouvent à droite de la virgule). Le type de données `NUMERIC(5,2)` permet de définir un tel nombre.

Le type de données DECIMAL

Le type de données `DECIMAL` est semblable au type de données `NUMERIC`. Il peut aussi disposer d'une partie décimale et vous pouvez spécifier sa précision et son échelle. La seule différence est que la précision que votre implémentation lui accorde peut être plus grande que celle que vous aurez spécifiée. Dans ce cas, c'est la plus grande précision qui sera utilisée. Si vous ne spécifiez ni précision ni échelle, l'implémentation utilise ses valeurs par défaut, exactement comme elle le fait pour le type `NUMERIC`.

Un élément que vous spécifiez comme étant un `NUMERIC (5,2)` ne peut jamais contenir un nombre dont la valeur est supérieure à 999,99. Une définition telle que `DECIMAL (5,2)` permet toujours d'enregistrer des nombres dont la valeur peut aller jusqu'à 999,99, mais aussi des nombres plus grands encore si l'implémentation le permet.

Utilisez le type `NUMERIC` ou `DECIMAL` si votre donnée comporte une partie décimale, et choisissez les types `SMALLINT` et `INTEGER` si la donnée est un nombre entier. Adoptez le type `NUMERIC` si vous voulez maximiser la portabilité, car une valeur dont vous définissez le type comme `NUMERIC (5,2)` occupera exactement la même place sur tous les systèmes.

Les numériques approximatifs

Certaines quantités peuvent prendre des valeurs dans une plage si grande qu'un ordinateur offrant une taille d'enregistrement fixe ne pourrait représenter ces valeurs exactement (cette taille peut être par exemple de 32 bits, 64 bits ou encore 128 bits). L'exactitude n'est alors plus nécessaire et une approximation devient acceptable. SQL définit trois types de données numériques approximatifs pour traiter ce genre de valeur.

Le type de données REAL

Le type de données REAL vous permet de stocker des nombres à virgule flottante en simple précision, cette dernière dépendant de l'implémentation, qui est elle-même liée au matériel que vous utilisez. Par exemple, la précision est (évidemment) plus importante sur une machine 64 bits que sur un système 32 bits.

Un nombre à virgule flottante est un nombre qui comporte une partie décimale. La partie décimale « flotte », c'est-à-dire qu'elle peut apparaître à n'importe quel endroit dans le nombre. 3,14 et 3,14159 sont des exemples de nombres à virgule flottante.

Le type de données DOUBLE PRECISION

Le type de données `DOUBLE PRECISION` vous permet de stocker un nombre à virgule flottante en double précision, cette dernière dépendant toujours de l'implémentation. La signification du mot `DOUBLE` dépend aussi de l'implémentation (étonnant, non ?). L'arithmétique en double précision est utilisée par les scientifiques. Différentes disciplines scientifiques ont divers besoins en terme de précision. Certaines implémentations de SQL s'adressent plus à une catégorie particulière d'utilisateurs, tandis que d'autres sont destinées à un public différent.

Sur certains systèmes, le type `DOUBLE PRECISION` occupe exactement deux fois plus de place que le type `REAL`, tant pour la mantisse que pour l'exposant. (Vous pouvez représenter n'importe quel nombre sous forme d'une mantisse multipliée par dix élevé à la puissance de l'exposant. Par exemple, vous pouvez écrire 6,626 sous la forme 6,626 E 3. Le

nombre 6,626 est la mantisse que vous multipliez par dix élevé à la puissance de l'exposant, c'est-à-dire 3 dans cet exemple.)

Vous ne tirez aucun bénéfice à représenter des nombres dont les valeurs sont très proches les unes des autres (par exemple 6,626 et 6,626001) à l'aide d'un type de données approximatif. Cependant, vous devez utiliser un type de données approximatif pour stocker les nombres dont la valeur est proche de zéro ou bien plus grande que un tel que 6626 E -34 (un très petit nombre). Les types numériques exacts ne permettent pas de stocker de tels nombres. D'autres systèmes représentent le type `DOUBLE PRECISION` à l'aide d'une mantisse deux fois plus grande et d'un exposant presque deux fois plus grand que la mantisse et l'exposant du type `REAL`. D'autres systèmes encore représentent le type `DOUBLE PRECISION` à l'aide d'une mantisse deux fois plus grande et d'un exposant ayant la même capacité que le type `REAL`. La précision est alors doublée, mais pas l'intervalle.

La spécification SQL n'essaie pas d'arbitrer les débats ou d'imposer la signification de `DOUBLE PRECISION`. Elle indique simplement que la précision d'un nombre `DOUBLE PRECISION` doit être plus grande que celle d'un `REAL`. Cette contrainte peut paraître quelque peu succincte, mais elle tient compte de la grande diversité des matériels que l'on peut rencontrer et représente donc probablement le meilleur compromis possible.

Le type de données FLOAT

Le type de données `FLOAT` est à utiliser si vous pensez faire un jour migrer votre base de données d'une plate-forme matérielle vers une autre dont la taille des registres est différente. Le type de données `FLOAT` vous permet de spécifier la précision, comme par exemple dans `FLOAT (5)`. Votre système utilisera la simple précision ou la double précision suivant ce que requiert ce que vous avez spécifié.

Utiliser le type `FLOAT` plutôt que les types `DECIMAL` ou `REAL` facilite le portage de votre base de données d'un matériel à un autre, car il vous permet de spécifier la précision. Les précisions de `DECIMAL` et de `FLOAT` dépendent pour leur part du matériel.

Si vous n'êtes pas certain du type numérique exact (NUMERIC/ DECIMAL) ou du type numérique approximatif (FLOAT/REAL) que vous devez utiliser, définissez toujours un type numérique exact. Les types de données numériques exacts consomment moins de ressources système et, comme il se doit, permettent de stocker des valeurs exactes (plutôt qu'approximatives). Vous devriez être en mesure de déterminer à l'avance si la plage des valeurs que peut prendre une donnée est telle que vous n'avez d'autre solution que d'utiliser un type de données numérique approximatif.

Les chaînes de caractères

Les bases de données permettent aujourd'hui de stocker des types de données très variés, dont des images, des sons et des animations. Bientôt ce sera peut-être les odeurs. Quel plaisir si vous pouviez humer la pizza que vous vous apprêtez à commander via Internet… Mais, dans la vraie vie d'aujourd'hui, les types de données les plus utilisés sont les nombres et les chaînes de caractères.

Il existe trois principaux types de données caractères :

- Les chaînes de longueur fixe (CHARACTERou CHAR).
- Les chaînes de longueur variable (CHARACTER VARYING ou VARCHAR).
- Les objets larges de type caractère (CHARACTER LARGE OBJECT ou CLOB).

Vous pouvez aussi disposer de trois variantes de ces types de données caractères :

- NATIONAL CHARACTER.
- NATIONAL CHARACTER VARYING.
- NATIONAL CHARACTER LARGE OBJECT.

Le type de données CHARACTER

Si vous définissez une colonne comme étant de type CHARACTER ou CHAR, vous pourrez spécifier le nombre de caractères qu'elle contient en utilisant la syntaxe CHARACTER(x), où x est la longueur de la chaîne. Par exemple, si vous indiquez que le type de données d'une colonne est CHARACTER (16), la longueur maximale des données que vous pourrez y saisir sera de 16 caractères. Si vous ne spécifiez aucun argument (c'est-à-dire, si vous ne fournissez aucune valeur à la place de x), SQL supposera que la longueur que vous souhaitez est de un (1) caractère.

Si vous saisissez une donnée dans un champ CHARACTER d'une longueur spécifiée, et que cette donnée comporte moins de caractères que n'en précise votre type, SQL remplira l'espace restant à l'aide de blancs.

Le type de données CHARACTER VARYING

Le type de données CHARACTER VARYING est utilisé si la taille des entrées d'une colonne peut varier, mais que vous ne voulez pas que SQL complète les chaînes à l'aide d'espaces. Ce type vous permet donc de stocker le nombre exact de caractères que l'utilisateur aura saisis. Il n'existe pas de valeur par défaut pour ce type de données. Pour le définir, utilisez la forme CHARACTER VARYING(x) ou VARCHAR(x), où x est le nombre maximal de caractères que vous autorisez.

Le type de données CHARACTER LARGE OBJECT

Le type de données CHARACTER LARGE OBJECT (CLOB) a été introduit dans SQL:1999. Comme son nom le laisse entendre, il est utilisé pour stocker des chaînes de caractères immenses, trop grandes pour le type CHARACTER. Les CLOB ressemblent à des chaînes de caractères classiques, mais leur utilisation est soumise à un certain nombre de restrictions.

Un CLOB ne peut être utilisé dans un prédicat PRIMARY KEY, FOREIGN KEY ou UNIQUE. De plus, il ne peut être utilisé dans une comparaison autre que l'égalité ou l'inégalité. Parce qu'ils sont très grands, les applications ne transfèrent généralement pas les CLOB dans une base de données. Un client spécial nommé localisateur CLOB est utilisé pour

manipuler les données CLOB. C'est un paramètre dont la valeur identifie un objet large de type caractère.

Un *prédicat* est une commande qui peut être soit vraie, soit fausse.

Les types de données NATIONAL CHARACTER, NATIONAL CHARACTER VARYING et NATIONAL CHARACTER LARGE OBJECT

Certains langages utilisent des caractères qui leur sont spécifiques. Par exemple, l'allemand (comme d'ailleurs le français) possède des caractères spéciaux qui n'existent pas en anglais. Quelques langages tels que le russe utilisent un jeu de caractères totalement différent de l'anglais. Même si l'anglais est le jeu de caractères par défaut de votre système, vous pourrez quand même utiliser des jeux de caractères étrangers, car les types de données NATIONAL CHARACTER, NATIONAL CHARACTER VARYING et NATIONAL CHARACTER LARGE OBJECT fonctionnent comme les types de données CHARACTER, CHARACTER VARYING et CHARACTER LARGE OBJECT, à ceci près que le jeu de caractères que vous spécifiez peut être différent du jeu de caractères défini sur votre système. Il est possible de spécifier le jeu de caractères quand vous définissez la colonne d'une table. Si vous le voulez, SQL vous permet d'attribuer un jeu de caractères différent à chaque colonne. Dans l'exemple suivant, nous créons une table qui utilise différents jeux de caractères :

```
CREATE TABLE TRADUCTION (
LANGAGE_1 CHARACTER (40),
LANGAGE_2 CHARACTER VARYING (40) CHARACTER SET
GREEK,
LANGAGE_3 NATIONAL CHARACTER (40),
LANGAGE_4 CHARACTER (40) CHARACTER SET KANJI
);
```

La colonne LANGAGE_1 contient des caractères du jeu de caractères par défaut de l'implémentation. La colonne LANGAGE_3 correspond aux caractères du jeu de caractères national de l'implémentation. La colonne LANGAGE_2 est destinée à contenir des caractères grecs. Et la colonne LANGAGE_4 contiendra des caractères kanji. Après une longue absence, les jeux de caractères asiatiques, tels que le kanji, sont de nouveau présents dans les SGBD.

Les chaînes binaires

Les types de données chaînes BINARY ont été introduits dans SQL:2008. Comme les données binaires sont fondamentales depuis l'ordinateur Atanasoff-Berry des années 1930, cette reconnaissance de leur importance semble assez tardive dans SQL (mais mieux vaut tard que jamais, je suppose). Il existe trois différents types de données binaires : BINARY, BINARY VARYING, et BINARY LARGE OBJECT.

Le type de données BINARY

Si vous définissez le type de données d'une colonne comme BINARY, vous pourrez spécifier le nombre d'octets que la colonne peut contenir en utilisant la syntaxe BINARY(x), où x est le nombre en question. Par exemple, si vous spécifiez BINARY (16), la chaîne binaire peut comprendre 16 octets. Les données binaires doivent être saisies octet par octet, en commençant par le premier octet.

Le type de données BINARY VARYING

Utilisez le type BINARY VARYING lorsque la longueur de la chaîne est variable. Pour spécifier ce type de données, utilisez la forme BINARY VARYING(x) ou VARBINARY(x), où x est le nombre maximum d'octets permis. La taille minimale de la chaîne binaire est zéro et sa taille maximale est x.

Le type de données BINARY LARGE OBJECT

Le type de données `BINARY LARGE OBJECT` (`BLOB`) est utilisé pour d'énormes chaînes binaires qui sont trop grandes pour le type `BINARY`. Les fichiers d'images et de musiques sont des exemples. Les `BLOB`s se comportent beaucoup comme des chaînes binaires, mais SQL impose un certain nombre de restrictions à leur manipulation.

En particulier, vous ne pouvez pas utiliser un `BLOB` comme prédicat `PRIMARY KEY`, `FOREIGN KEY` ou `UNIQUE`. De plus, il est impossible de comparer des `BLOB`s autrement que pour l'égalité ou l'inégalité. Les `BLOB`s sont énormes, si bien que les applications ne les transfèrent pas d'une base de données. Elles utilisent un type de données client particulier nommé *localisateur de BLOB* pour manipuler les données d'un `BLOB`. Ce localisateur est un paramètre dont la valeur identifier le gros objet binaire.

Les booléens

Le type de données `BOOLEAN` comprend les valeurs vrai (`true`), faux (`false`) et inconnu (`unknown`). Si un booléen dont la valeur est vrai ou faux est comparé à `NULL` ou à la valeur inconnu, le résultat sera la valeur *inconnu*.

Les dates et les heures

Le standard SQL définit cinq types de données différents pour gérer les dates et les heures. Cependant ces types se recoupent tant les uns les autres que certaines implémentations ne les gèrent pas tous.

Les implémentations qui ne supportent pas les cinq types de données pour les dates et les heures peuvent vous poser des problèmes si vous tentez de migrer vos bases de données sur un autre système. Si cela arrive, vérifiez que les implémentations source et destination représentent les dates et les heures de la même manière.

Le type de données DATE

Le type de données `DATE` mémorise l'année, le mois et le jour, dans cet ordre. L'année est un nombre de quatre chiffres, le mois et le jour sont des

nombres à deux chiffres. Une valeur `DATE` peut représenter n'importe quelle date depuis l'année 0001 jusqu'à l'année 9999. La longueur de `DATE` est de 10 positions, comme dans 1957-08-14.

Le type de données TIME WITHOUT TIME ZONE

Le type de données `TIME WITHOUT TIME ZONE` mémorise les heures, les minutes et les secondes. Les heures et les minutes utilisent chacune deux chiffres. Les secondes peuvent occuper deux chiffres, mais elles peuvent aussi comporter une partie décimale. Il est donc possible de représenter 9 heures 32 minutes et 58,436 secondes sous la forme 09 : 32 : 58.436.

La précision de la partie décimale dépend de l'implémentation, mais elle est d'au moins six chiffres. Une valeur `TIME WITHOUT TIME ZONE` occupe huit positions (en comprenant les deux-points) quand la valeur n'a aucune partie décimale ou neuf positions (en comprenant le séparateur décimal) plus la taille de la partie décimale quand la valeur en comporte une. Vous pouvez spécifier le type de données `TIME WITHOUT TIME ZONE` comme `TIME`, ce qui correspond à une heure sans partie décimale, ou comme `TIME WITHOUT TIME ZONE (p)`, où *p* est le nombre de chiffres qu'occupe la partie décimale. Dans l'exemple précédent, le type de données utilisé est `TIME WITHOUT TIME ZONE (3)`.

Le type de données TIMESTAMP WITHOUT TIME ZONE

Le type de données `TIMESTAMP WITHOUT TIME ZONE` stocke la date et l'heure. Les longueurs ainsi que les restrictions appliquées aux valeurs des données `TIMESTAMP WITHOUT TIME ZONE` sont les mêmes que pour `DATE` et `TIME WITHOUT TIME ZONE`, à une différence près : la longueur par défaut de la partie décimale de l'heure d'une `TIMESTAMP WITHOUT TIME ZONE` est de six chiffres et non de zéro.

Si la valeur n'a pas de partie décimale, la longueur de `TIMESTAMP WITHOUT TIME ZONE` est de 19 positions : dix positions pour la date,

un espace qui tient lieu de séparateur et huit positions, dans cet ordre. Si la valeur a une partie décimale (six chiffres par défaut), la longueur est de 20 positions plus le nombre de chiffres de la partie décimale. La vingtième position est occupée par le séparateur décimal. Vous spécifiez le type d'un champ comme `TIMESTAMP WITHOUT TIME ZONE` en utilisant soit `TIMESTAMP WITHOUT TIME ZONE` soit `TIMESTAMP WITHOUT TIME ZONE (p)`, où *p* est le nombre de chiffres de la partie décimale. La valeur de *p* ne peut être négative, et l'implémentation détermine sa valeur maximale.

Le type de données TIME WITH TIME ZONE

Le type de données `TIME WITH TIME ZONE` est identique au type `TIME WITHOUT TIME ZONE`, à ceci près qu'il contient une information supplémentaire : un décalage horaire par rapport au temps universel (autrement dit par rapport au méridien de Greenwich ou GMT). La valeur de ce décalage peut aller de – 12 : 59 à + 13 : 00. Cette information supplémentaire occupe six positions après le temps : un séparateur, un signe plus ou moins, puis le décalage en heures (deux chiffres) et en minutes (deux chiffres) séparées par un deux-points. Une valeur `TIME WITH TIME ZONE` sans partie décimale (par défaut) occupe 14 positions. Si vous spécifiez une partie décimale, la longueur est de 15 positions plus le nombre de chiffres que vous aurez alloué à cette partie décimale.

Le type de données TIMESTAMP WITH TIME ZONE

Le type de données `TIMESTAMP WITH TIME ZONE` fonctionne exactement comme le type `TIMESTAMP WITHOUT TIME ZONE`, à ceci près qu'il contient lui aussi un décalage horaire. Cette information supplémentaire occupe six positions. La longueur totale d'une `TIMESTAMP WITH TIME ZONE` passe donc à 25 positions sans partie décimale et à 26 positions plus le nombre de chiffres alloués à la partie décimale si elle en comporte.

Les intervalles

Les types de données intervalles sont apparentés aux types de données date et heure. Un *intervalle* est la différence entre deux valeurs date et heure. Dans de nombreuses applications qui manipulent des dates, des heures ou les deux à la fois, vous avez parfois besoin de déterminer l'intervalle qui sépare deux dates ou deux heures.

SQL reconnaît deux types distincts d'intervalles : année-mois et jour-heure. Le premier représente l'écart en années et mois entre deux dates. Le second correspond à la différence en jours, heures, minutes et secondes entre deux instants d'un même mois. Il n'est pas possible de mélanger les deux dans un calcul, car la longueur des mois varie (28, 29, 30 ou 31 jours).

Le type XML

Le type de données prédéfini XML est le petit dernier de la famille. Les données XML ont une structure arborescente, ce qui signifie qu'un nœud racine peut avoir plusieurs nœuds enfants, chacun étant à son tour susceptible de posséder sa propre descendance. Introduit dans SQL:2003, le type XML a été enrichi dans SQL/XML : 2005. L'édition 2005 définit cinq sous-types paramétrés en plus du type d'origine. Les valeurs XML peuvent être des instances de deux ou plusieurs types, car certains sous-types sont eux-mêmes des sous-types d'autres sous-types (peut-être faudrait-il les désigner par sous-sous-types, voire sous-sous-sous-types ; fort heureusement, SQL:2008 définit un standard pour se référer aux sous-types).

Les modificateurs primaires du type XML sont `SEQUENCE`, `CONTENT` et `DOCUMENT`. Les modificateurs secondaires sont `UNTYPED`, `ANY` et `XMLSCHEMA`. La Figure 2.1 montre la structure arborescente qui illustre les relations entre les sous-types.

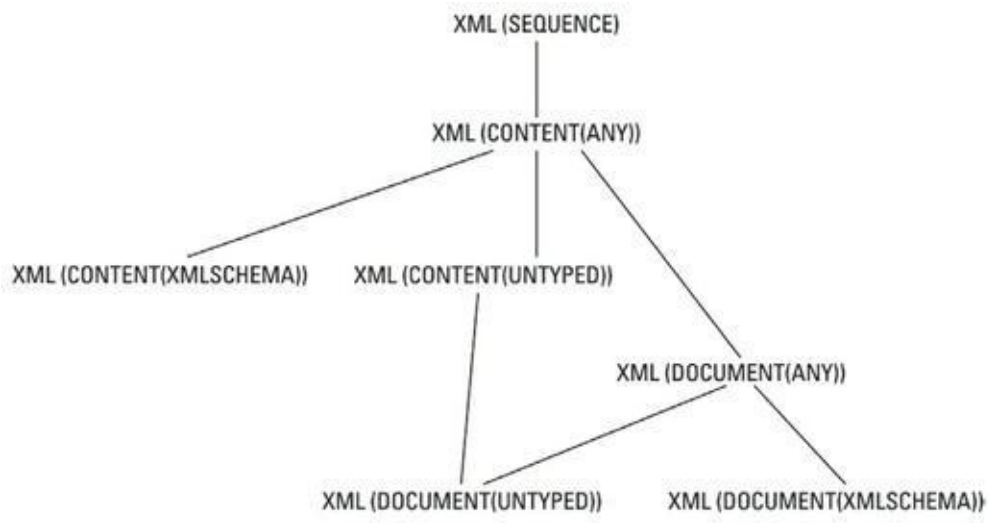


FIGURE 2.1 Les relations entre les sous-types XML.

Voyons brièvement les types XML que vous vous devez de connaître. Commençons par les types de base pour terminer par les plus complexes :

- `XML(SEQUENCE)` : Toute valeur XML est soit une valeur SQL NULL, soit une séquence XQuery. De cette manière, chaque valeur XML est une instance du type `XML(SEQUENCE)`. *XQuery* est un langage de requête spécifiquement conçu pour extraire des informations à partir de données XML. Il s'agit du type XML le plus basique.

- `XML(SEQUENCE)` est le moins restrictif des types XML. Il peut accepter des valeurs qui ne respectent pas totalement les canons XML. Les autres types sont, quant à eux, sensiblement moins conciliants.
- `XML(CONTENT(ANY))` : Il s'agit d'un type légèrement plus restrictif que `XML(SEQUENCE)`. Chaque valeur XML qui est soit un NULL, soit un nœud de document XQuery (ou un enfant de ce nœud) est une instance de ce type. Toute instance de `XML(CONTENT(ANY))` est également une instance de `XML(SEQUENCE)`. Les valeurs du type

`XML(CONTENT(ANY))` ne sont pas nécessairement conformes aux canons XML. Il peut s'agir des résultats intermédiaires d'une requête qui seront par la suite remis en bonne et due forme.

» `XML(CONTENT(UNTYPED))` : Ce type est encore plus restrictif que le précédent. Par conséquent, toute valeur du type `XML(CONTENT(UNTYPED))` est aussi une instance des types `XML(CONTENT(ANY))` et `XML(SEQUENCE)`. Toute valeur XML qui est soit la valeur nulle, soit une valeur non nulle de type `XML(CONTENT(ANY))`, est un nœud d'un document XQuery D, tel que ce qui suit est vrai pour tout nœud élément contenu dans l'arbre T dont la racine est D :

- La propriété `type-name` est `xdt : untyped`.
- La propriété `nilled` est `False`.
- Pour tout nœud attribut contenu dans T, la propriété `name` est `xdt : untypedAtomic`.
- Pour tout nœud attribut contenu dans T, la propriété `type` est une valeur de type `XML(CONTENT(UNTYPED))`.

» `XML- (CONTENT(XMLSCHEMA))` : C'est le second sous-type de `XML(CONTENT(ANY))` avec `XML(CONTENT(UNTYPED))`. Toute valeur XML qui est soit la valeur nulle, soit une valeur non nulle de type `XML(CONTENT(ANY))` est aussi un nœud d'un document XQuery D, tel que tout nœud élément contenu dans l'arbre T de racine D :

- Est valide conformément au schéma XML S, ou

- Est valide conformément à l'espace de noms N dans le schéma S, ou
- Est valide conformément à la déclaration de schéma E dans un schéma XML S, ou
- Est une valeur de type `XML(CONTENT(XMLSCHEMA))`, dont le descripteur de type comprend celui du schéma S et, si N est spécifié, l'URI de l'espace de noms de N, ou si E est spécifié, l'URI de l'espace de noms de E et le `NCName` de E.

» `XML(DOCUMENT(ANY))` : Il s'agit d'une autre variante du type `XML(CONTENT(ANY))` avec une restriction supplémentaire : une instance de `XML(DOCUMENT(ANY))` est un nœud de document possédant exactement un élément enfant, zéro ou plusieurs nœuds commentaires, et zéro ou plusieurs nœuds d'instructions de traitement XQuery.

» `XML(DOCUMENT(UNTYPED))` : Chaque valeur qui est soit la valeur `NULL`, soit une valeur non nulle de type `XML(CONTENT(UNTYPED))` qui se trouve dans un nœud document dont la propriété `children` comprend exactement un nœud élément, zéro ou plusieurs nœuds commentaires, et zéro ou plusieurs instructions de traitement XQuery, est une valeur de type `XML(DOCUMENT(UNTYPED))`. Toutes les instances de `XML(DOCUMENT(UNTYPED))` sont également des instances du type `XML(CONTENT(UNTYPED))`. Ce sont également des instances du type `XML(DOCUMENT(ANY))`.

> `XML(DOCUMENT(UNTYPED))` est le plus restrictif de tous les sous-types. En d'autres termes, les restrictions des autres sous-types s'appliquent également à lui. Tout document qualifié comme étant un `XML(DOCUMENT(UNTYPED))` est aussi une instance de tous les autres sous-types XML.

Les types ROW

Le type de données `ROW` a été introduit dans SQL:1999. Il n'est pas très facile de comprendre son usage, et il se peut qu'en tant qu'utilisateur novice ou intermédiaire de SQL vous n'ayez jamais à vous en servir. Après tout, les gens s'en sont bien passés entre 1986 et 1999.

Une des principales caractéristiques du type de données ROW est qu'il viole les règles de normalisation énoncées par E.F. Codd à l'aube des bases de données relationnelles. Je reviendrai sur ces règles dans le Chapitre 5. L'une des caractéristiques de la première forme normale est qu'un champ d'une ligne d'une table ne peut contenir plusieurs valeurs. Un champ doit donc recevoir une et une seule valeur. Cependant, le type de données `ROW` vous permet de déclarer qu'une ligne de données tout entière sera contenue dans un seul champ d'une seule ligne. En d'autres termes, il est possible d'insérer une ligne dans une ligne.

Les *formes normales*, selon le Dr Codd, définissent les caractéristiques des bases de données relationnelles. L'inclusion du type `ROW` dans le standard SQL a été la première tentative d'extension du modèle relationnel pur.

Considérez l'instruction SQL suivante, qui déclare l'adresse d'une personne sous la forme d'un type `ROW` :

```
CREATE ROW TYPE type_adresse (
Rue CHARACTER VARYING (25)
Ville CHARACTER VARYING(20)
Etat CHARACTER (2)
Code_Postal CHARACTER VARYING (9)
);
```

Une fois défini, le nouveau type ROW peut être utilisé dans la définition d'une table :

```
CREATSSSE TABLE CLIENTS (
ClientID INTEGER PRIMARY KEY,
Prénom CHARACTER VARYING (25),
Nom CHARACTER VARYING (20),
Adresse type_adresse
Téléphone CHARACTER VARYING (15)
);
```

Tout l'intérêt est ici de pouvoir définir une et une seule fois ce qu'est une adresse et de pouvoir réutiliser cette définition dans la déclaration de diverses tables : table de clients, table des employés, etc.

Les types Collection

Une fois le processus de transgression des règles intangibles de SQL entamé avec SQL:1999, la définition de types violant la première forme normale est devenue possible. Un champ peut donc dès lors contenir une pleine collection d'objets et non plus une seule et unique entité. Le type ARRAY a été introduit avec SQL:1999, et le type MULTISET a été ajouté dans SQL:2003.

Deux collections ne peuvent être comparées que si elles ont le même type (soit ARRAY, soit MULTISET) et si les types de leurs éléments sont comparables. Du fait que les tableaux comportent un nombre défini d'entrées, il est possible de comparer les éléments correspondants. Ce n'est pas le cas des MULTISET, mais la comparaison reste faisable si (a) une énumération existe pour chaque multiset concerné, et si (b) les énumérations peuvent être appariées.

Le type ARRAY

Comme le type ROW, le type de données ARRAY viole lui aussi la première forme normale (1NF), mais d'une manière différente. Le type

ARRAY est une collection, pas un type distinct comme le sont les types CHARACTER ou NUMERIC. Pour l'essentiel, un type ARRAY permet à un autre type de prendre plusieurs valeurs qui seront mémorisées dans un seul champ d'une table.

Supposons par exemple que votre entreprise désire pouvoir contacter ses clients à tout instant, qu'ils soient au travail, à leur domicile ou sur la route. Vous souhaiteriez donc stocker plusieurs numéros de téléphone pour chaque personne. Vous pouvez le faire en déclarant l'attribut Téléphone sous la forme d'un tableau de la manière suivante :

```
CREATE TABLE CLIENTS (
ClientID INTEGER PRIMARY KEY,

Nom CHARACTER VARYING (25),
Prénom CHARACTER VARYING (20),
Adresse type_adresse
Téléphone CHARACTER VARYING (15)
);
```

La notation ARRAY[3] vous permet de stocker jusqu'à trois numéros de téléphone dans la table CLIENTS. Ce type de données enfreint les restrictions imposées par le Dr Codd quand il énonça les règles de normalisation en préférant l'intégrité des données à la flexibilité de leur gestion. SQL:1999 a rendu la gestion des données plus souple mais rend aussi leur structure plus complexe.

Plus la structure devient complexe, plus vous devez prendre garde que son utilisation remette en cause l'intégrité de votre base de données. Il est nécessaire de mesurer avec précision les effets de chaque action que vous effectuez sur votre base de données. Un tableau est ordonné dans le sens où chacun de ses éléments est associé avec exactement une position ordinale dans ledit tableau.

Si la cardinalité d'un tableau est inférieure au maximum déclaré, les cellules inutilisées dans le tableau sont considérées comme inexistantes. Elles ne sont pas considérée comme contentant des valeurs nulles ; elles n'existent tout simplement pas.

Vous pouvez accéder aux éléments spécifiques d'un tableau en faisant figurer leur index entre crochets. Si vous avez un tableau nommé Téléphone, alors Téléphone[3] vous permet de vous référer au troisième élément figurant dans ce tableau.

Depuis SQL:1999, il est possible de trouver la cardinalité d'un tableau en invoquant la fonction `CARDINALITY`. Depuis SQL:2011, vous pouvez découvrir la cardinalité maximale d'un tableau en utilisant la fonction `ARRAY_MAX_CARDINALITY`. Cette fonction est très utile, car elle vous permet d'écrire des routines de portée générale qui s'appliquent à des tableaux de cardinalités maximales différentes. Les routines où la cardinalité maximale est codée en dur ne s'appliquent qu'aux tableaux de la cardinalité maximale correspondante, et doivent donc être réécrites pour s'appliquer à des tableaux de cardinalités maximales différentes.

Alors que SQL:1999 a introduit le type de données `ARRAY` et la possibilité d'adresser les éléments spécifiques d'un tableau, rien n'a été prévu pour supprimer des éléments du tableau. Cet oubli a été corrigé dans SQL:2011 avec l'introduction de la fonction `TRIM_ARRAY`, qui vous permet de supprimer des éléments à la fin d'un tableau.

Le type MULTISET

Un multiset est une collection non ordonnée. Des éléments spécifiques du multiset peuvent ne pas être référencés, car ils ne sont affectés à aucune position ordinale particulière.

Les types REF

Les types `REF` ne font pas partie du noyau de SQL. Cela signifie qu'un SGBD peut se proclamer compatible avec le standard SQL sans implémenter les types `REF`. Il ne s'agit pas d'un type de données distinct comme le sont `CHARACTER` et `NUMERIC`. C'est un pointeur vers une donnée, un type de données ou un type de données abstrait qui réside dans une ligne d'une table (un site). Déréférencer le pointeur permet de retrouver la valeur stockée dans le site cible.

Si vous n'y comprenez rien, ne vous inquiétez pas, car vous n'êtes pas le seul. L'utilisation des types `REF` requiert une connaissance avancée des principes de la programmation orientée objet. Ce livre se gardera bien de

plonger dans les arcanes de ce sujet. En fait, puisque les types REF ne font pas partie du cœur de SQL, vous feriez mieux de vous en passer. Si vous souhaitez rester compatible avec un maximum de plates-formes de SGBD, tenez-vous-en aux fonctionnalités du noyau de SQL.

Les types définis par l'utilisateur

Les types définis par l'utilisateur (UDT, pour User-Defined Types) ont été introduits avec SQL:1999. Ils viennent du monde de la programmation orientée objet (POO). En tant que programmeur SQL, vous n'êtes plus limité par les types de données définis dans la spécification standard. Vous pouvez définir vos propres types de données en vous basant sur les principes des types de données abstraites (ADT, pour Abstract Data Types) courants dans les langages orientés objet tels que le C++.

L'un des principaux intérêts des UDT est qu'ils peuvent être utilisés pour gérer le problème de compatibilité entre SQL et le langage « emballé » autour de SQL. Un problème récurrent de SQL est que ses types de données prédéfinis ne sont pas forcément les mêmes que ceux employés par les langages dans lesquels des instructions SQL sont intégrées. Grâce aux UDT, le programmeur d'une base de données peut créer dans SQL des types de données qui correspondent exactement aux types du langage qu'il utilise.

Un UDT encapsule des attributs et des méthodes. Le monde extérieur voit les attributs et les résultats produits par les méthodes, mais l'implémentation spécifique de ces méthodes lui est cachée. Il est possible d'imposer des restrictions sur l'accès aux attributs et aux méthodes d'un UDT en spécifiant qu'ils sont publics, privés ou protégés.

- **Les attributs (ou les méthodes) publics** sont accessibles à tous les utilisateurs de l'UDT.
- **Les attributs (ou les méthodes) privés** ne sont accessibles qu'à l'UDT lui-même.
- **Les attributs (ou les méthodes) protégés** ne sont accessibles qu'à l'UDT lui-même ou à ses types dérivés.

Vous voyez qu'un UDT de SQL se comporte pour l'essentiel comme une classe dans un langage de programmation orienté objet. Il existe deux

formes de types définis par l'utilisateur : les types distincts et les types structurés.

Les types distincts

Les types distincts sont les plus élémentaires des types définis par l'utilisateur. Ils prennent la forme d'un type de donnée unique et sont construits à partir d'un des types prédéfinis (appelés dans ce cas types source). Plusieurs types distincts qui sont tous basés sur un même type source sont différents les uns des autres et ne sont donc pas directement comparables. Par exemple, vous pouvez utiliser des types distincts pour représenter diverses monnaies :

```
CREATE DISTINCT TYPE DollarUS AS DECIMAL (9,2);
```

crée un nouveau type de données pour les dollars, basé sur le type prédéfini `DECIMAL`. Vous pouvez ensuite ajouter un autre type distinct de la manière suivante :

```
CREATE DISTINCT TYPE Euro AS DECIMAL (9,2);
```

Il est maintenant possible de définir des tables qui utilisent ces types :

```
CREATE TABLE FactureUS (
FacID INTEGER PRIMARY KEY,

ClientID INTEGER,
EmpID INTEGER,
TotalVente DollarUS,
Taxe DollarUS,
Port DollarUS,
GrandTotal DollarUS
);

CREATE TABLE FactureEuro (
FacID INTEGER PRIMARY KEY,
ClientID INTEGER,
```

```
EmpID INTEGER,
TotalVente Euro,
Taxe Euro,
Port Euro,
GrandTotal Euro
);
```

Le type `DollarUS` et le type `Euros` sont tous les deux basés sur le type `DECIMAL`, mais leurs instances ne peuvent être directement comparées entre elles ni avec des instances du type `DECIMAL`. Dans la réalité, il est possible de convertir en SQL des dollars US en euros à l'aide d'un opérateur spécial (`CAST`). Une fois la conversion effectuée, la comparaison devient possible.

Types structurés

La seconde forme de type défini par l'utilisation (le type structuré) est exprimée à l'aide d'une liste d'attributs et de méthodes au lieu d'être basée sur un unique type source prédéfini.

Constructeurs

Lorsque vous créez un UDT structuré, le SGBD lui associe automatiquement une fonction dite *constructeur* qui prend le même nom que cet UDT. Le travail du constructeur consiste à initialiser les attributs de l'UDT à leur valeur par défaut.

Mutateurs et accesseurs

Toujours lors de la création d'un UDT structuré, le SGBD ajoute tout aussi automatiquement deux autres fonctions : un mutateur et un accesseur. Un *mutateur*, lorsqu'il est invoqué, change la valeur d'un attribut du type structuré correspondant. À l'inverse, un *accesseur* sert à retrouver la valeur d'un attribut dans un type structuré. Vous pouvez inclure des accesseurs dans des instructions SELECT afin de récupérer des valeurs dans une base de données.

Sous-types et supertypes

Il peut exister une relation hiérarchique entre deux types structurés. Prenons un exemple. Un type appelé `MusiqueCDudt` possède un sous-type dénommé RockCDudt et un autre dénommé ClassiqueCDudt. Par rapport à ces deux styles musicaux, MusiqueCDudt est leur supertype. `RockCDudt` est un sous-type propre de `MusiqueCDudt` s'il n'existe aucun intermédiaire entre eux, autrement dit quelque chose qui soit à la fois sous-type de MusiqueCDudt et supertype de RockCDudt. Si, maintenant, on crée un sous-type de `RockCDudt` appelé `HeavyMetalCDudt`, ce dernier sera par héritage un sous-type de `MusiqueCDudt`, mais pas un sous-type propre. Vous suivez toujours ?

Exemple de type structuré

Vous pouvez créer des UDT structurés de la manière suivante :

```
/* Crée un UDT appelé MusiqueCDudt */
CREATE TYPE MusiqueCDudt AS
/* Spécifie les attributs */
Titre CHAR(40),
Cout DECIMAL(9,2),
PrixConseille DECIMAL(9,2),
/* Les sous-types sont autorisés */
NOT FINAL ;

CREATE TYPE RockCDudt UNDER MusiqueCDudt NOT
FINAL;
```

Le sous-type `RockCDudt` hérite des attributs du supertype `MusiqueCDudt`.

```
CREATE TYPE HeavyMetalCDudt UNDER RockCDudt
FINAL;
```

Maintenant que vous avez les types, il est possible de créer les tables qui les utilisent. Par exemple :

```
CREATE TABLE METALSKU (
```

```
Album HeavyMetalCDudt,
SKU INTEGER);
```

Il n'y a plus qu'à ajouter des lignes à la table :

```
BEGIN
/* Déclare une variable temporaire a */
DECLARE a = HeavyMetalCDudt ;
/* Exécute la fonction constructeur */
SET a = HeavyMetalCDidt() ;
/* Exécute la première fonction mutateur */
SET a = a.Titre('Edouard the Great') ;
/* Exécute la deuxième fonction mutateur */
SET a = a.Cout(7.5) ;
/* Exécute la troisième fonction mutateur */
SET a = a.PrixConseille(15.99) ;

INSERT INTO METALSKU VALUES (a, 31415926);
END
```

Les types définis par l'utilisateur pour les types Collection

Dans la section précédente « Types distincts », je vous ai montré comme créer un type défini par l'utilisateur à partir d'un type prédéfini, en utilisant l'exemple de la création du type USDollar à partir du type Euros. Cette possibilité a été introduite dans SQL:1999. SQL:2011 l'étend en vous permettant de créer un nouveau type défini par l'utilisateur à partir d'un type Collection. Cela permet au développeur de définir des méthodes s'appliquant à un tableau comme un tout, et non plus à chacun de ses éléments, comme l'imposait SQL:1999.

Liste des types de données

Le Tableau 2.2 contient la liste des divers types de données, chacun étant accompagné d'un exemple.

TABLEAU 2.2 Les types de données.

Type de données	Exemple
CHARACTER (20)	'Radio Amateur'
VARCHAR (20)	'Radio Amateur'
CLOB (1000000)	'Cette chaîne de caractères fait un million de caractères… '
SMALLINT, BIGINT ou INTEGER	7500
NUMERIC ou DECIMAL	3425.432
REAL, FLOAT ou DOUBLE PRECISION	6.626E-34
BLOB (1000000)	'1001001110101011010101010101… '
BOOLEAN	'true'
DATE	DATE '1957-08-14'
TIME (2) WITHOUT TIME ZONE1	TIME '12 : 46 : 02.43' WITHOUT TIME ZONE
TIME (3) WITH TIME ZONE	TIME '12 : 46 : 02.432-08 : 00' WITH TIME ZONE
TIMESTAMP WITHOUT TIME ZONE (0) TIMESTAMP	TIMESTAMP '1957-08-14 12 : 46 : 02' WITHOUT TIME ZONE
Type de données	Exemple

TIMESTAMP WITH TIME ZONE (0) TIMESTAMP	TIMESTAMP '1957-08-14 12 : 46 : 02-08 : 00' WITH TIME ZONE
INTERVAL DAY	INTERVAL '4' DAY
XML(SEQUENCE)	<Client>Allen G. Taylor</Client>
ROW	ROW (Rue VARCHAR (25), Ville VARCHAR (20), Etat CHAR (2), Code_Postal VARCHAR (9))
ARRAY	INTEGER ARRAY [15]
MULTISET	Aucun littéral ne s'applique au type MULTISET
REF	Pas un type mais un pointeur
USER DEFINED TYPE	Type de monnaie basé sur DECIMAL

L'argument spécifie le nombre de chiffres de la partie décimale.

Votre implémentation spécifique de SQL peut ne pas supporter tous les types de données qui viennent d'être décrits dans cette section. De plus, il est possible que votre implémentation utilise des types de données non standard que je ne décris pas ici.

Les valeurs nulles

Si un champ d'une base de données contient une donnée, ce champ a une valeur spécifique. Un champ qui ne contient aucune donnée est dit avoir une valeur nulle (NULL). N'oubliez pas que :

- Dans un champ numérique, une valeur nulle n'a pas la même signification que zéro.
- Dans un champ caractère, une valeur nulle n'est pas un caractère espace.

Le zéro numérique et le caractère espace sont des valeurs définies. Une valeur nulle indique que la valeur du champ n'est pas déterminée.

Un champ peut prendre une valeur nulle dans de nombreux cas. Voici quelques exemples :

- **La valeur existe mais vous ne la connaissez pas encore**. Vous passez `MASSE` à nul dans la ligne `Top` de la table `QUARK` avant que la masse du quark ne soit déterminée.
- **La valeur n'existe pas encore**. Vous définissez `TOTAL_VENDU` comme étant nul dans la ligne SQL pour Les Nuls de la table `LIVRES`, car vous ne savez pas encore combien d'exemplaires ont été vendus.
- **Le champ n'est pas utilisable pour cette ligne particulière**. Vous affectez à `SEXE` une valeur nulle dans la rangée `Z6PO` de la table `EMPLOYES`, car `Z6PO` est un droïde.
- **La valeur est hors de portée**. Vous attribuez la valeur nulle à `RICHESSE` dans la rangée `BILL GATES` de la table `FORTUNES`, car le maximum possible de 999 999 999 euros est largement dépassé.

Un champ peut prendre une valeur nulle pour de nombreuses raisons. Ne tirez pas des conclusions hâtives sur la signification d'une valeur nulle.

Les contraintes

Les contraintes sont des restrictions que vous appliquez sur les données que quelqu'un peut saisir dans la base de données. Par exemple, si les entrées d'une certaine colonne numérique doivent appartenir à une certaine plage de valeurs. Si quelqu'un tente de saisir une valeur qui n'appartient pas à cet

intervalle, la base de données risque d'être contaminée. Vous pouvez prévenir cette nuisance en appliquant une contrainte de plage à la colonne.

Généralement, un programme qui utilise une base de données applique des contraintes à cette base. Cependant, les SGBD les plus récents vous permettent d'imposer directement des contraintes à vos bases. Cela présente plusieurs avantages. Si plusieurs applications utilisent la même base de données, vous n'aurez à appliquer les contraintes qu'une seule fois. De plus, ajouter des contraintes au niveau de la base de données est toujours plus simple que de les intégrer dans une application. Dans de nombreux cas, il suffit d'ajouter une clause à votre instruction CREATE.

Je décrirai en détail les contraintes et les assertions (qui sont des contraintes que vous appliquez à plusieurs tables) dans le Chapitre 5.

Utiliser SQL sur un système client/serveur

SQL est un sous-langage de données qui fonctionne sur un système isolé ou sur un système multi-utilisateur. SQL est particulièrement efficace sur les systèmes client/serveur. Sur de tels systèmes, les utilisateurs de plusieurs machines clients qui se connectent à la machine serveur peuvent accéder à une base de données au travers d'un réseau local (LAN) ou de tout autre moyen de communication. Le programme qui fonctionne sur la machine client contient les commandes SQL de manipulation de données. La portion du SGBD qui réside côté client envoie les commandes au serveur via le moyen de communication qui les relie. Sur l'autre versant, la portion serveur du SGBD interprète et exécute les commandes SQL, puis renvoie les résultats au client. Vous pouvez encoder des opérations très complexes en SQL sur le client, puis les faire décoder et exécuter sur le serveur. Cela permet de surcroît d'économiser de la bande passante.

Si vous récupérez les données en utilisant SQL sur un système client/serveur, seules les données que vous demandez transitent via le support de communication qui relie le client au serveur. À l'inverse, sur un système se contentant de partager des ressources et qui n'utilise qu'un serveur doté d'une intelligence minimale, des volumes de données énormes doivent circuler sur le réseau pour que vous puissiez récupérer les données dont vous avez besoin. Il est inutile de préciser que ce genre d'opérations ralentit considérablement les opérations (tout en accroissant les failles dans

la sécurité). L'architecture client/serveur combinée à SQL permet d'obtenir de bonnes performances sur tous les types de réseaux.

Le serveur

Tant qu'il ne reçoit pas de requête du client, le serveur ne fait rien. Il ne fait qu'attendre. Si plusieurs clients lui présentent des requêtes simultanément, le serveur doit y répondre immédiatement. Il diffère des machines clients en ce qu'il dispose d'un vaste espace de stockage disque. Le serveur est optimisé pour accéder rapidement aux données. Et comme il doit gérer beaucoup de requêtes simultanées, il a besoin d'un processeur rapide, voire même de plusieurs processeurs.

Ce qu'est le serveur

Le *serveur* (abréviation pour *serveur de base de données*) est la partie d'un système client/serveur qui contient la base de données. On y trouve aussi le côté serveur du SGBD. Cette partie du SGBD interprète les commandes émanant des clients et les traduit en opérations sur la base de données. Le logiciel serveur formate aussi les résultats des requêtes et renvoie les résultats au client qui l'a demandé.

Ce que fait le serveur

Le travail du serveur est relativement simple et évident. Tout ce qu'un serveur doit faire est de lire, interpréter et exécuter les commandes que des clients lui envoient par le réseau. Ces commandes sont rédigées dans l'un des nombreux sous-langages de données.

Un sous-langage ne définit pas complètement un langage : il en implémente seulement une partie. Un sous-langage de données ne traite que de la gestion des données. Ce sous-langage contient des opérations pour insérer, mettre à jour, supprimer et sélectionner des données, mais il ne dispose pas d'instructions de contrôle de flux telles que les boucles DO, les variables locales, les fonctions, les procédures ou les opérations d'entrée/sortie sur les imprimantes. SQL est le plus courant des sous-langages de données. Les autres sous-langages de données propriétaires ont été supplantés par SQL sur tous les types de machines. Depuis SQL:1999, SQL a acquis nombre des fonctionnalités qui manquent aux sous-langages de données traditionnels.

Cependant, il n'est pas encore devenu un langage de programmation généraliste à part entière, et il doit pour cette raison être combiné à un langage hôte pour réaliser des applications de bases de données.

Le client

La partie *client* d'un système client/serveur consiste en un composant matériel et un composant logiciel. Le composant matériel est l'ordinateur client et son interface au réseau local. Le client matériel peut parfaitement être identique au matériel serveur. Ce qui distingue le client du serveur, c'est le logiciel.

Ce qu'est le client

Le rôle principal du client consiste à fournir une interface utilisateur. Pour autant que l'utilisateur est concerné, la machine client est l'ordinateur et l'interface utilisateur est l'application. L'utilisateur peut ne même pas réaliser qu'un processus se déroule sur un serveur. Le serveur est généralement hors de sa vue (dans une autre pièce, dans un autre bâtiment, à l'autre bout de la planète…). Hormis l'interface utilisateur, le client contient aussi le programme d'application et la partie client du SGBD. Le programme d'application effectue les tâches spécifiques que vous lui demandez, comme saisir des fiches de paye ou des commandes. La partie client du SGBD exécute les commandes de ce programme et échange des données et des commandes SQL de manipulation de données avec le côté serveur du SGBD.

Ce que fait le client

Le client affiche l'information à l'écran et répond aux informations que l'utilisateur transmet via le clavier, la souris ou tout autre périphérique. Le client peut aussi traiter des données qui proviennent d'autres postes du réseau (quelle que soit par ailleurs la nature physique de celui-ci). La partie client du SGBD effectue tout le travail de « réflexion ». C'est donc celle qui intéresse le plus le développeur. La partie serveur ne fait que traiter les requêtes que lui présente le client d'une manière mécanique, répétitive.

Utiliser SQL sur l'Internet ou un

intranet

La gestion des bases de données via Internet ou un intranet est radicalement différente de la gestion via un système client/ serveur classique. Tout d'abord, sur un système client/serveur traditionnel, l'essentiel des fonctionnalités du SGBD réside sur la machine client. Sur un système basé sur Internet, l'essentiel du SGBD se trouve sur le serveur. Le client peut n'héberger rien d'autre qu'un navigateur Web. Au mieux, il contient une extension du navigateur, telle qu'un plug-in pour Netscape ou un contrôle ActiveX. Par conséquent, la « masse critique » du système se trouve reportée sur le serveur. Cela présente plusieurs avantages :

- La portion client du système (le navigateur) est peu onéreuse.
- Vous disposez d'une interface utilisateur standardisée.
- La maintenance du client est facile.
- La relation client/serveur est standardisée.
- Vous possédez un support tout à fait courant pour afficher des données multimédias.

Les principaux inconvénients que présente la manipulation d'une base de données via Internet sont la sécurité et l'intégrité des données :

- Pour protéger l'information de toute altération ou consultation par un tiers non autorisé, le serveur Web et le client doivent supporter une technique puissante de cryptage.
- Les navigateurs ne procèdent pas à de véritables contrôles de validation des données saisies.
- Les tables de la base de données qui résident sur plusieurs serveurs peuvent se désynchroniser.

Les extensions client et serveur conçues pour gérer ces problèmes ont fait d'Internet une architecture pour laquelle il est possible de développer des applications de base de données. L'architecture d'un intranet ressemble à celle d'Internet, mais la sécurité pose un moindre problème. Comme l'organisation qui assure la maintenance de l'intranet dispose d'un contrôle physique sur les machines clients, les serveurs et le réseau qui les relie, un intranet est bien moins exposé aux attaques des pirates. Cependant, les problèmes du contrôle des informations saisies et de la désynchronisation demeurent.

Chapitre 3
Les composants de SQL

DANS CE CHAPITRE :

- **Créer des bases de données.**
- **Manipuler des données.**
- **Protéger les bases de données.**

SQL est un langage conçu pour la création et la maintenance des données dans des bases relationnelles. Quoique les vendeurs de systèmes de gestion de bases de données relationnelles utilisent leur propre version de SQL, un standard ISO/ANSI (révisé en 2011) définit ce qu'est SQL. Toutes les implémentations s'écartent plus ou moins de cette norme. Par conséquent, s'en tenir au standard est la meilleure tactique pour faire fonctionner une base de données sur plus d'une plate-forme.

Bien que SQL ne soit pas vraiment un langage de programmation, il est quand même doté de quelques outils impressionnants. Trois langages pour le prix d'un vous permettent de créer, de modifier, de maintenir et de sécuriser une base de données relationnelle :

- **Le langage de définition de données (DDL, pour Data Definition Language) :** C'est la partie de SQL que vous utilisez pour créer (définir complètement) une base de données, modifier sa structure et la détruire quand vous n'en avez plus besoin.
- **Le langage de manipulation de données (DML, pour Data Manipulation Language) :** Il permet la maintenance d'une base de données. En utilisant ce puissant outil, vous

pouvez spécifier exactement ce que vous voulez faire avec les données de votre base de données : les entrer, les modifier ou les extraire.

- **Le langage de contrôle de données (DCL, pour Data Control Language) :** Une armure pour protéger vos données. Quand il est utilisé correctement, DCL permet de sécuriser votre base de données ; la qualité de cette protection dépend de l'implémentation. Si votre implémentation ne fournit pas une protection suffisante, vous devez hausser le niveau de celle-ci dans votre programme d'application.

Ce chapitre présente DDL, DML et DCL.

Le langage de définition de données (DDL)

Le langage de définition de données (DDL) est une partie de SQL que vous utilisez pour créer, modifier ou détruire les éléments de base (c'est le cas de le dire) d'une base de données relationnelle. Il s'agit des tables, des vues, des schémas, des catalogues, des clusters et éventuellement d'autres choses. Dans cette section, je présente la hiérarchie qui lie tous ces éléments ainsi que les commandes qui permettent de les manipuler.

Dans le Chapitre 1, j'ai parlé des tables et des schémas en précisant qu'un schéma est une structure générale qui contient des tables. Les tables et les schémas sont donc deux éléments de la hiérarchie du contenu d'une base de données relationnelle. Vous pouvez décomposer cette hiérarchie ainsi :

- Les tables contiennent des lignes et des colonnes.
- Les schémas contiennent des tables et des vues.
- Les catalogues contiennent des schémas.

La base de données elle-même contient des catalogues. Vous verrez qu'on désigne parfois la base de données par le terme *cluster*.

« Y a qu'à ! » est un mauvais conseil

Supposons que vous soyez chargé de créer une base de données pour votre société. Excité à l'idée de créer une structure utile, valable, bien conçue et de grande importance pour le futur, vous prenez place devant votre ordinateur et commencez à saisir des commandes SQL CREATE. Correct ?

Eh bien non ! C'est en fait la meilleure solution pour aboutir au désastre. De nombreux projets de développement de bases de données partent dans le mur dès le départ parce que l'excitation l'emporte sur une planification rigoureuse. Même si vous avez une idée très précise de la structure de votre base de données, vous devez tout écrire noir sur blanc avant de toucher le clavier de votre ordinateur.

Le développement de bases de données peut assez bien être comparé au jeu d'échecs. Dans une compétition d'échecs, vous trouvez un déplacement qui vous paraît bon. La pendule tourne, la tension fait son œuvre, et l'urgence de jouer ce coup vous étreint. Mais il y a une forte probabilité pour que vous ayez manqué quelque chose. Les GM (Grands Maîtres) conseillent souvent aux débutants de s'asseoir sur leurs mains. Et ce n'est qu'à moitié une plaisanterie. Si placer vos mains sous vos cuisses vous évite de jouer un coup mal calculé, alors c'est le bon geste à faire. Si vous prenez un peu plus de temps pour étudier la position, vous pourrez trouver un coup encore meilleur (ou découvrir l'attaque foudroyante que vous réserve votre adversaire). De la même manière, plonger dans la création d'une base de données sans une réflexion préalable suffisante conduit presque toujours à une structure qui, dans le meilleur des cas, sera mal optimisée. Et conduit droit au désastre dans le pire des cas (comme une invitation à la corruption des données lancée aux quatre vents). Certes, s'asseoir sur vos mains ne vous aidera pas à avancer. Prendre un stylo dans une de ces mains et commencer à planifier sur papier le schéma de votre base de données, si.

La liste suivante recense quelques procédures que vous devez conserver à l'esprit quand vous planifiez votre base de données :

- Identifiez toutes les tables.
- Définissez les colonnes que chaque table doit contenir.

- Attribuez une *clé primaire* unique à chaque table (je parle des clés dans les Chapitres 4 et 5).
- Assurez-vous que chaque table de la base de données partage au moins une colonne avec une autre table. Ces colonnes partagées servent à établir des liens logiques entre l'information contenue dans une table et une information correspondante dans une autre table.
- Passez chaque table dans la troisième forme normale (3NF) ou plus pour prévenir toute anomalie pouvant être engendrée par une insertion, une suppression ou une mise à jour (je traite de la normalisation des bases de données dans le Chapitre 5).

Quand vous aurez terminé votre conception sur le papier, vous pourrez la transférer sur l'ordinateur à l'aide de la commande SQL CREATE. Plus vraisemblablement, vous utiliserez l'interface graphique utilisateur (GUI) du SGBD pour cela. Si vous utilisez votre GUI, votre SGBD convertira toutes vos actions en SQL « dans les coulisses ».

Créer des tables

Une table d'une base de données est un tableau à deux dimensions composé de lignes et de colonnes. Vous pouvez créer une table en utilisant la commande SQL CREATE TABLE. Vous spécifiez en paramètre le nom et le type de données de chaque colonne.

Une fois que vous avez créé la table, vous allez y stocker des données (ce qui est une fonction de DML et non de DDL). Si vos besoins évoluent, vous pouvez modifier la structure de la table en utilisant la commande ALTER TABLE. S'il arrive un jour que la table ne soit plus utile, il est possible de l'éliminer en utilisant la commande DROP. Les différentes formes des commandes CREATE et ALTER, ainsi que la commande DROP, constituent le langage de définition de données de SQL.

Imaginons que vous êtes le concepteur d'une base de données et que vous ne voulez plus que vos tables dégénèrent à cause de mises à jour répétées. Vous décidez alors de structurer les tables de votre base de données selon la meilleure forme normale pour préserver l'intégrité des données.

La *normalisation* (c'est à elle seule un vaste sujet d'études) est une manière de structurer les tables d'une base de données pour que les mises à jour n'introduisent pas d'anomalies. Chaque table que vous créez contient des colonnes qui correspondent à des attributs qui sont étroitement liés entre eux.

Par exemple, vous pouvez créer une table CLIENTS avec les attributs `CLIENT_ID`, `NOM`, `PRENOM`, `RUE`, `VILLE`, `ETAT`, `CODE_POS-TAL` et `TELEPHONE`. Tous ces attributs sont plus étroitement liés à l'entité client qu'à toute autre entité d'une base de données qui contient parfois de nombreuses tables. Ces attributs représentent toutes les informations permanentes que vous conservez sur vos clients.

La plupart des systèmes de gestion de bases de données disposent d'un outil graphique pour créer des tables de bases de données. Cependant, vous pouvez aussi créer des tables en utilisant une commande SQL. L'exemple suivant vous montre comment créer une table CLIENTS de cette manière :

```
CREATE TABLE CLIENTS (
CLIENT_ID INTEGER NOT NULL,
PRENOM CHARACTER (15),
NOM CHARACTER (20) NOT NULL,
RUE CHARACTER (25),
VILLE CHARACTER (20),
ETAT CHARACTER (2),
CODE_POSTAL INTEGER,
TELEPHONE CHARACTER (13) ) ;
```

Vous précisez pour chaque colonne son nom (par exemple, `CLIENT_ID`), son type de données (par exemple, `INTEGER`) et éventuellement une ou plusieurs contraintes (par exemple, `NOT NULL`).

La Figure 3.1 vous montre une portion de la table CLIENTS remplie de quelques données.

CLIENT_ID	PRENOM	NOM	RUE	VILLE	ETAT	CODEPOSTAL	TELEPHONE
1	Harold	Percival	26262 S. Howa	Westminster	CA	92683	619-555-1234
2	Jerry	Appel	32323 S. River	Santa Anna	CA	92705	207-555-2345
3	Adrian	Hansen	232 Glenwood	Hollis	NH	3049	603-555-3456
4	John	Baker	2222 Lafayette	Garden Grove	CA	92643	309-555-4567
5	Michael	Penn	77730 S. New E	Irvine	CA	92715	714-555-5678
6	Bob	Michimoto	25252 S. Kelm	Stanton	CA	92610	612-555-6789
7	Linda	Smith	444 S.E. Seve	Hudson	NH	3051	811-555-7891
8	Robert	Funnell	2424 Sheri Cou	Anaheim	CA	92640	603-555-8912
9	Bill	Checkal	9595 Curry Dr	Stanton	CA	92610	521-555-9123
10	Jed	Style	3535 Randall S	Santa Anna	CA	92705	706-555-4321
(NuméroAuto)						0	

FIGURE 3.1 La table CLIENTS créée en utilisant la commande CREATE TABLE.

Si l'implémentation de SQL que vous utilisez ne répond pas complètement au standard ANSI/ISO, la syntaxe que vous devrez utiliser peut différer de celle que j'utilise dans ce livre. Consultez la documentation de votre SGBD pour plus d'informations à ce sujet.

Chambre avec vue

Un jour, vous aurez besoin de récupérer des données de la table CLIENTS. Vous ne voudrez pas visualiser tout, mais simplement quelques colonnes et quelques lignes. Vous aurez alors besoin d'une vue.

Une vue est une table virtuelle. Sur la plupart des implémentations, une vue n'a aucune existence physique indépendante. La définition de la vue existe dans les métadonnées de la base, mais ses données proviennent de la table ou des tables dont vous dérivez cette vue. Les données de la vue ne sont donc pas physiquement dupliquées quelque part sur le disque. Certaines vues contiennent des colonnes et des lignes spécifiques d'une seule table. D'autres, les *vues multitables*, proposent des données qui proviennent de plusieurs tables.

Vue avec une seule table

La réponse à vos questions se trouve parfois dans une seule table de votre base de données. Dans ce cas, vous pouvez créer une vue simple. Par exemple, supposons que vous désiriez consulter les noms et les numéros de téléphone de tous les clients américains qui vivent dans l'État du New Hampshire. Vous créez une vue à partir de la table CLIENTS en utilisant la commande SQL suivante :

```
CREATE VIEW NH_CLIENT AS
```

```
SELECT CLIENTS.PRENOM,
CLIENTS.NOM,
CLIENTS.TELEPHONE

FROM CLIENTS
WHERE CLIENTS.ETAT = ‘NH’ ;
```

La Figure 3.2 vous montre comment la vue est dérivée de la table CLIENTS.

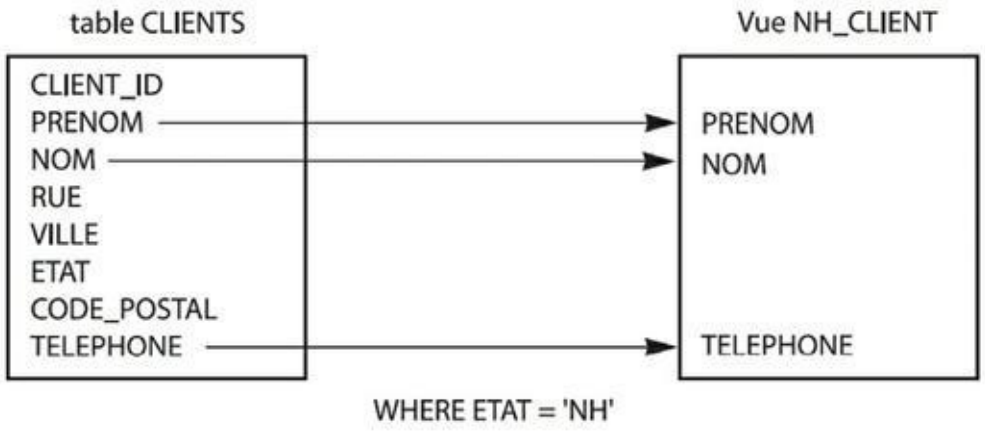


FIGURE 3.2 Vous dérivez la vue NH_ CLIENT de la table CLIENTS.

Ce code est tout à fait correct, mais vous pourriez arriver au même résultat plus rapidement si l’implémentation de SQL que vous utilisez présumait que toutes les références aux tables sont faites derrière la clause FROM. Dans ce cas, vous pouvez réduire la commande à ces quelques lignes :

```
CREATE VIEW NH_CLIENT AS
SELECT PRENOM, NOM, TELEPHONE
FROM CLIENTS
WHERE ETAT = ‘NH’;
```

Quoique la seconde syntaxe soit plus agréable à écrire et à lire, elle est plus sensible aux effets d’une commande `ALTER TABLE`. Ces effets seront négligeables dans l’exemple que nous venons de voir, car notre vue n’utilise pas de jointure. Cependant, les vues qui utilisent JOIN sont moins robustes si vous n’utilisez pas des noms totalement qualifiés. Je traite des jointures au Chapitre 11.

Créer une vue multitable

Il arrive souvent que vous deviez extraire des données de deux ou plusieurs tables pour répondre à une question. Par exemple, supposons que vous travailliez pour un magasin de sport et que vous désiriez obtenir la liste des clients qui ont acheté un équipement de ski l'année dernière pour leur envoyer un mailing promotionnel. Vous aurez besoin d'informations contenues dans les tables CLIENT, PRODUIT, FACTURE et LIGNE_FACTURE. Vous pouvez créer une vue multitable qui contienne toutes ces informations. Une fois que vous aurez créé cette vue, elle tiendra compte des mises à jour effectuées sur les tables depuis la dernière fois où vous l'aurez consultée.

La base de données du magasin de sport contient quatre tables : CLIENT, PRODUITS, FACTURE et LIGNE_FACTURE. Ces tables sont structurées comme le montre le Tableau 3.1.

TABLEAU 3.1 Les tables du magasin de sport.

Table	Colonne	Type de données	Contrainte
CLIENT	CLIENT_ID	INTEGER	NOT NULL
	PRENOM	CHARACTER (15)	
	NOM	CHARACTER (20)	NOT NULL
	RUE	CHARACTER (25)	
	VILLE	CHARACTER (20)	
	ETAT	CHARACTER (2)	
	CODE_POSTAL	INTEGER	
	TELEPHONE	CHARACTER (13)	
PRODUIT	PRODUIT_ID	INTEGER	NOT NULL
	NOM	CHARACTER (25)	
	DESCRIPTION	CHARACTER (30)	

	CATEGORIE	CHARACTER (15)	
	VENDEUR_ID	INTEGER	
	NOM_VENDEUR	CHARACTER (30)	
FACTURE	NUMERO_FACTURE	INTEGER L	NOT NUL
	CLIENT_ID	INTEGER	
	DATE_FACTURE	DATE	
	TOTAL_VENTE	NUMERIC (9,2)	
	TOTAL_REMIS	NUMERIC (9,2)	
	FORME_PAIEMENT	CHARACTER (10)	
LIGNE_FACTURE	NUMERO_LIGNE	INTEGER	NOT NULL
	NUMERO_FACTURE	INTEGER	
	PRODUIT_ID	INTEGER	
	QUANTITE	INTEGER	
	PRIX_VENTE	NUMERIC (9,2)	

Notez que quelques colonnes du Tableau 3.1 contiennent la contrainte NOT NULL. Ces colonnes sont soit des clés primaires de leur table, soit des colonnes dont vous ne voulez pas qu'elles contiennent une valeur nulle. La clé primaire d'une table identifie de manière unique chaque ligne. Elle doit contenir une valeur non nulle. Je traite des clés dans le Chapitre 5.

Les tables sont reliées entre elles par des colonnes qu'elles possèdent en commun. La Figure 3.3 illustre ces relations. Elles sont décrites ci-dessous.

- La table CLIENT entretient une relation « un à plusieurs » avec la table FACTURE. Un client peut effectuer plusieurs achats, ce qui génère autant de factures. Cependant, chaque facture n'est liée qu'à un unique client.

- La table FACTURE entretient une relation « un à plusieurs » avec la table LIGNE_FACTURE. Une facture peut contenir plusieurs lignes, mais chaque ligne n'apparaît que sur une et une seule facture.
- La table PRODUIT entretient aussi une relation « un à plusieurs » avec la table LIGNE_FACTURE. Un produit peut apparaître dans plusieurs lignes sur une ou plusieurs factures. Cependant, chaque ligne ne peut correspondre qu'à un unique produit.

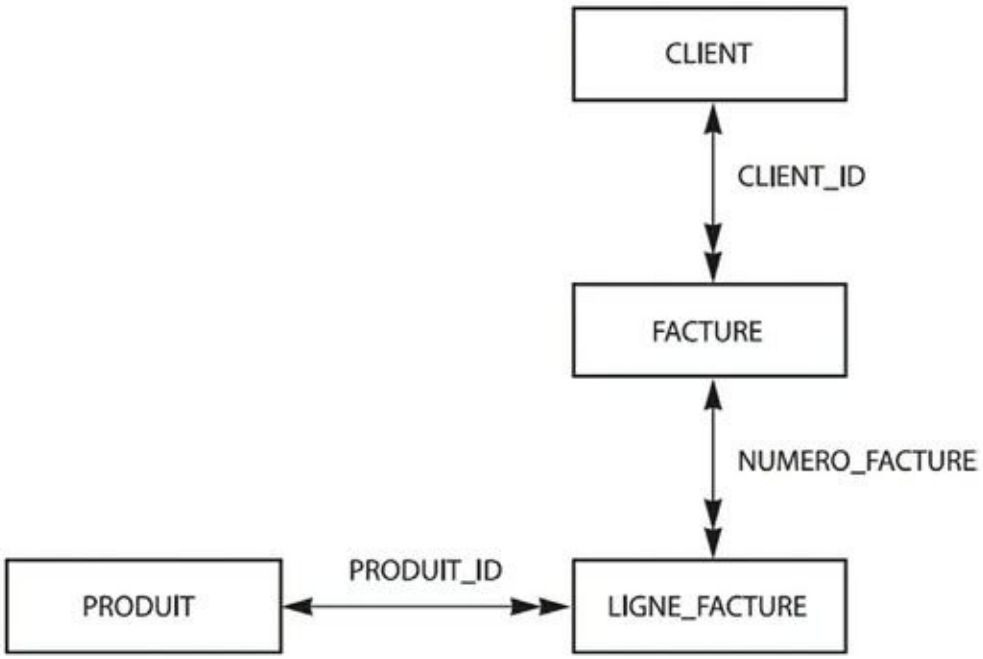


FIGURE 3.3 La structure de la base de données du magasin de sport.

- La table CLIENT est reliée à la table FACTURE par la colonne `CLIENT_ID`. La table FACTURE est reliée à la table LIGNE_FACTURE par la colonne NUMERO_FACTURE. La table PRODUIT est reliée à la table LIGNE_FACTURE par la colonne `PRODUIT_ID`.

Ces liens sont l'essence d'une base de données relationnelle.

Pour obtenir l'information que vous recherchez sur vos clients, vous aurez besoin de NOM, PRENOM, RUE, VILLE, ETAT et CODE_POS-TAL dans la table CLIENT, de CATEGORIE dans la table PRODUIT et de NUMERO_FACTURE dans la LIGNE_FACTURE. Vous pouvez créer la vue en utilisant les commandes suivantes :

```
CREATE VIEW SKI_CLIENT1 AS
SELECT NOM,
          PRENOM,
          RUE,
          VILLE,
          ETAT,
          CODE_POSTAL,
          NUMERO_FACTURE
FROM CLIENT JOIN FACTURE
USING (CLIENT_ID) ;
CREATE VIEW SKI_CLIENT2 AS
SELECT NOM,
          PRENOM,
          RUE,
          VILLE,
          ETAT,
          CODE_POSTAL,
          PRODUIT_ID
FROM SKI_CLIENT1 JOIN LIGNE_FACTURE
USING (NUMERO_FACTURE) ;
CREATE VIEW SKI_CLIENT3 AS
SELECT NOM,
          PRENOM,
          RUE,
          VILLE,
          ETAT,
          CODE_POSTAL,
          CATEGORIE
```

```
FROM SKI_CLIENT2 JOIN PRODUIT
USING (PRODUIT_ID) ;
CREATE VIEW SKI_CLIENT AS
SELECT DISTINCT NOM,
          PRENOM,
          RUE,
          VILLE,
          ETAT,
          CODE_POSTAL,
FROM SKI_CLIENT3
WHERE CATEGORIE = ‘Ski’ ;
```

Ces instructions `CREATE VIEW` combinent les données des différentes tables en utilisant l’opérateur `JOIN`. La Figure 3.4 et les notes qui suivent détaillent ce processus.

- La première instruction combine les colonnes de la table CLIENT avec une colonne de la table FACTURE pour créer la vue `SKI_CLIENT1`.
- La deuxième instruction combine les colonnes de la table `SKI_CLIENT1` avec une colonne de la table FACTURE_LIGNE pour créer la vue `SKI_CLIENT2`.
- La troisième instruction combine les colonnes de la table `SKI_CLIENT2` avec une colonne de la table PRODUIT pour créer la vue `SKI_CLIENT3`.

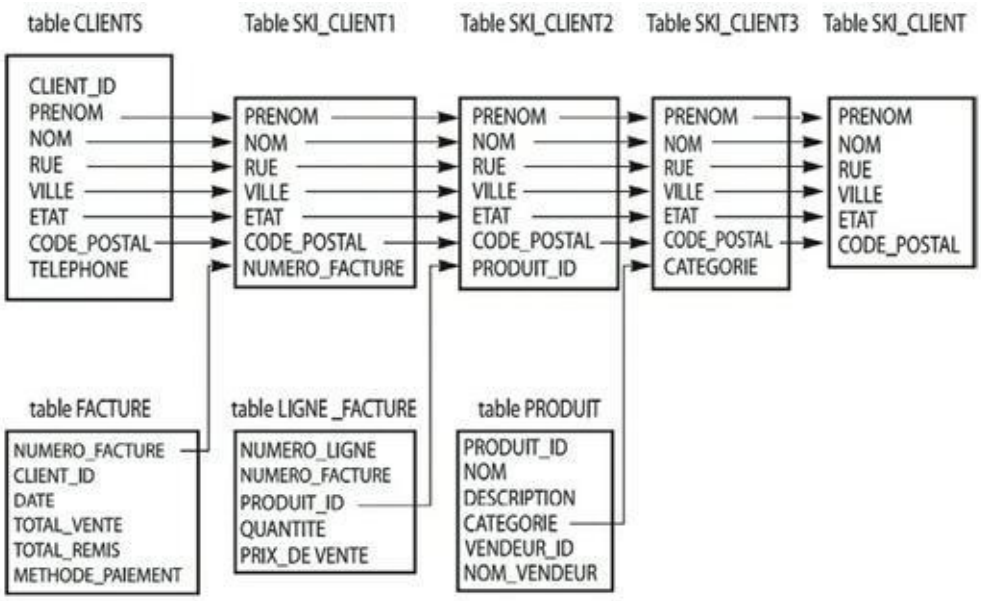


FIGURE 3.4 La création d'une vue multitable en utilisant des jointures.

» La quatrième instruction supprime toutes les lignes qui n'entrent pas dans la catégorie « ski ». Le résultat final est une vue (`SKI_CLIENT`) qui contient les noms et les adresses de tous les clients qui ont acheté au moins un produit de la catégorie « ski ».

Le mot clé `DISTINCT` dans la clause `SELECT` de la quatrième instruction `CREATE VIEW` vous garantit que vous n'aurez qu'une et une seule entrée pour chaque client, même si certains clients ont acheté plusieurs articles de ski.

Il est possible de créer une vue multitable en une seule instruction SQL. Toutefois, si vous pensez que l'une ou l'autre des commandes précédentes sont complexes, imaginez quel degré de complexité atteindrait celle qui cumulerait toutes leurs fonctions. Je préfère la simplicité à la complexité, si bien que chaque fois que c'est possible, je choisis le moyen le plus simple d'accomplir une fonction, même si ce n'est pas très « efficient ».

Organiser les tables en schémas

Une table est un ensemble de lignes et de colonnes qui traite généralement d'entités d'un type spécifique telles que les clients, les produits ou les factures. Dans la réalité, il faut parfois manipuler des informations liées à plusieurs de ces entités. Sur le plan organisationnel, vous rassemblez les tables que vous associez à ces entités pour former un *schéma logique* (un schéma logique est la structure organisationnelle qui correspond à un ensemble de tables liées entre elles).

En plus des schémas logiques, la base de données contient des *schémas physiques*. Le schéma physique est la manière dont la donnée et ses objets associés, tels que des index, sont physiquement organisés sur les périphériques de stockage du système. Quand je mentionne le schéma d'une base de données, je me réfère au schéma logique et non au schéma physique.

Dans un système où plusieurs projets indépendants peuvent coexister, vous pouvez assigner toutes les tables qui sont reliées entre elles à un seul schéma. Les autres groupes de tables seront le cas échéant placés dans leurs propres schémas.

Vous devriez nommer les schémas pour vous assurer que personne ne mélangera accidentellement les tables d'un projet avec celles d'un autre projet. Chaque projet dispose de son propre schéma associé, que vous distinguez des autres par son nom. Cependant, il est fréquent que des tables de différents projets portent le même nom (par exemple CLIENT ou PRODUIT). Pour lever toute ambiguïté, vous devriez désigner vos tables dans vos instructions SQL en les préfixant par le nom de leur schéma (comme dans `NOM_SCHEMA. TABLE_NOM`). Si vous ne fournissez aucun nom de schéma, SQL considère que la table appartient au schéma par défaut.

Ordonner par catalogue

Utiliser plusieurs schémas peut se révéler insuffisant quand le système de gestion de bases de données est très volumineux. En effet, sur un environnement distribué de base de données que manipulent de nombreux utilisateurs, vous pouvez rencontrer des schémas qui portent le même nom. Pour éviter que ce problème ne se pose, SQL a ajouté un élément supplémentaire dans la hiérarchie des conteneurs : le catalogue. Un catalogue est un ensemble nommé de schémas.

Vous pouvez *qualifier* le nom d'une table en utilisant le nom de son catalogue et le nom de son schéma. Cela vous garantit que la table ne sera pas confondue avec une table homonyme d'un schéma homonyme. Le nom de table ainsi qualifié prendra la forme suivante :

```
NOM_CATALOG.NOM_SCHEMA.NOM_TABLE
```

La hiérarchie des conteneurs d'une base de données comprend à son sommet les clusters, mais rares sont les systèmes qui en requièrent l'utilisation. La plupart du temps, les choses s'arrêtent au catalogue. Un catalogue regroupe des schémas, un schéma regroupe des tables et des vues, lesquelles sont constituées de colonnes et de lignes.

Le catalogue inclut aussi *l'information de schéma*. Cette dernière contient les tables système. Les tables système gèrent les métadonnées associées aux autres schémas. Dans le Chapitre 1, j'ai défini une base de données comme une collection d'enregistrements intégrés capable de se décrire elle-même. Ce sont les métadonnées contenues dans les tables système qui rendent cette description possible.

Du fait que les catalogues sont identifiés par leur nom, il est possible d'en utiliser plusieurs dans une base de données. Chaque catalogue peut contenir plusieurs schémas, et chaque schéma peut contenir à son tour plusieurs tables. Et, bien entendu, chaque table est susceptible de posséder de multiples colonnes et de multiples lignes. Ces relations hiérarchiques sont représentées sur la Figure 3.5.

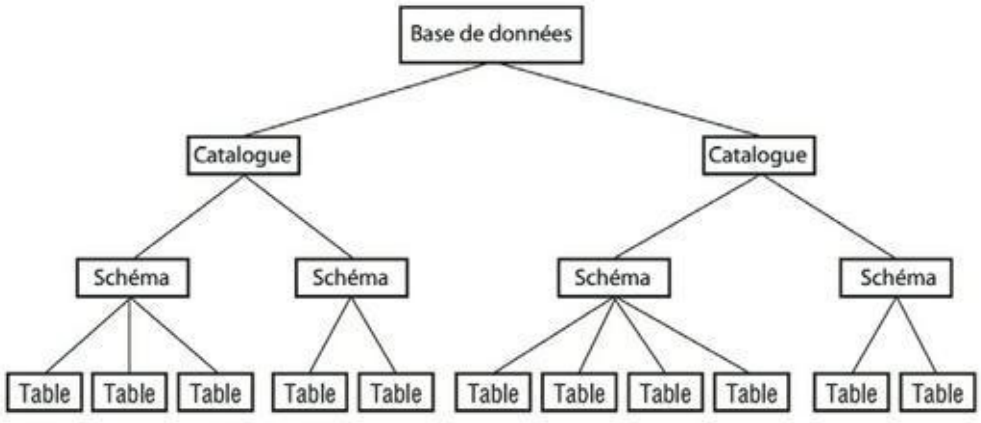


FIGURE 3.5 La structure hiérarchique d'une base de données SQL.

Se familiariser avec les commandes

DDL

Le langage de définition de données (DDL) permet de manipuler la structure de la base de données alors que DML, le langage de manipulation de données, permet de traiter les données stockées dans cette structure. Sous SQL, DDL est composé de trois commandes :

- `CREATE` : Vous utilisez les diverses formes de la commande `CREATE` pour créer les structures essentielles de la base de données.
- `ALTER` : Vous utilisez la commande `ALTER` pour modifier les structures que vous avez créées.
- `DROP` : Si vous utilisez la commande `DROP` sur une table, la commande détruit non seulement les données de la table mais aussi sa structure.

Dans les sections suivantes, je vais vous donner une description rapide des commandes de DDL. Nous nous servirons de ces commandes dans les exemples des Chapitres 4 et 5.

CREATE

La commande CREATE vous permet de créer de nombreux objets de SQL dont les schémas, les domaines, les tables et les vues. En utilisant l'instruction `CREATE SCHEMA`, vous pouvez créer un schéma, identifier son propriétaire et spécifier son jeu de caractères par défaut. Par exemple :

```
CREATE SCHEMA VENTES
AUTHORIZATION RESPONSABLE_VENTES
DEFAULT CHARACTER SET ASCII_FULL ;
```

Utilisez l'instruction `CREATE DOMAIN` pour appliquer des contraintes à des valeurs de colonne ou pour spécifier un ordre de regroupement. Les contraintes que vous appliquez au domaine déterminent la nature des objets que ce dernier a le droit de contenir. Vous pouvez créer des domaines après avoir défini un schéma. Par exemple :

```
CREATE DOMAIN Age AS INTEGER
CHECK (AGE > 20) ;
```

Pour les tables, vous disposez de l'instruction `CREATE TABLE`. Et l'instruction `CREATE VIEW` sert comme il se doit à créer des vues. Ci-avant dans ce chapitre, je vous ai présenté des exemples d'utilisation de ces deux instructions. Quand vous utilisez CREATE TABLE pour créer une nouvelle table, vous pouvez simultanément spécifier des contraintes sur les colonnes de cette dernière.

Cependant, il arrive que vous deviez définir des contraintes qui ne s'appliquent pas à une table en particulier mais à tout le schéma. Servez-vous alors de l'instruction `CREATE ASSERTION` pour définir ces contraintes.

Vous disposez aussi des instructions `CREATE CHARACTER`, `CREATE COLLATION` et `CREATE TRANSLATION` pour créer des jeux de caractères, des séquences d'interclassement et des tables de traduction. (Une séquence d'interclassement définit l'ordre dans lequel vous effectuez des comparaisons et des tris. Les tables de traduction contrôlent la conversion de chaînes de caractères d'un jeu de caractères à un autre.)

ALTER

Une fois que vous aurez créé une table, il se peut que vous ayez à la modifier. L'instruction `ALTER TABLE` vous servira à ajouter, modifier et supprimer des colonnes dans cette table. ALTER s'utilise aussi sur des domaines.

DROP

Il est très facile de supprimer une table du schéma d'une base de données. Utilisez simplement la commande `DROP TABLE`. Cela supprime toutes les données de la table ainsi que les métadonnées qui définissent cette dernière dans le dictionnaire des données.

Le langage de manipulation de

données (DML)

Comme je l'ai dit au début de ce chapitre, DDL est la partie de SQL qui permet de créer, modifier et détruire des structures de la base de données. DDL ne manipule pas les données elles-mêmes. Ce rôle est dévolu au langage de manipulation de données (DML). Quelques instructions de DML sont faciles à comprendre, d'autres semblent souvent très complexes. En effet, si une instruction DML comprend plusieurs expressions, clauses, prédicats ou sous-requêtes, il peut devenir très difficile de comprendre ce qu'elle signifie. Il est alors indispensable de découper l'instruction en morceaux et d'étudier chacun d'entre eux.

Les instructions DML dont vous disposez sont INSERT, UPDATE, DELETE et SELECT. Ces instructions peuvent comprendre plusieurs parties, dont de multiples clauses. Chaque clause peut à son tour contenir des expressions de valeurs, des connecteurs logiques, des prédicats, des fonctions d'agrégation et des sous-requêtes. Toutes ces clauses vous permettent de mieux extraire l'information de la base de données en discriminant précisément les enregistrements qu'elle contient. Dans le Chapitre 6, je traite de l'utilisation des commandes de DML. Je rentrerai dans le détail de celles-ci dans les Chapitres 8 à 14.

Les expressions de valeurs

Vous pouvez utiliser des *expressions de valeurs* pour combiner deux ou plusieurs valeurs. Il existe neuf sortes d'expressions de valeurs, chacune correspondant à un type de données précis :

- Numérique.
- Chaîne.
- Date et heure.
- Intervalle.
- Booléen.
- Défini par l'utilisateur.
- Ligne.

» Collection.

Les types Booléen, Défini par l'utilisateur et Ligne ont été introduits dans SQL:1999. Certaines implémentations ne les gèrent pas encore. Si vous voulez utiliser ces types de données, vérifiez ce qu'en dit votre implémentation.

Les expressions de valeurs numériques

Pour combiner des valeurs numériques, utilisez les opérateurs d'addition (+), de soustraction (-), de multiplication (*) et de division (/). Les lignes suivantes contiennent quelques exemples de telles expressions :

```
12 – 7
15/3 - 4
6 * (8 + 2)
```

Les valeurs qui figurent dans cet exemple sont des littéraux numériques. Ces valeurs pourraient tout aussi bien être des noms de colonnes, des paramètres, des variables hôtes ou des sous-requêtes. L'essentiel est qu'il s'agisse toujours de valeurs numériques. En voici quelques exemples :

```
SOUSTOTAL + TAXE + PORT
6 * KM/HEURE
:mois/12
```

Le deux-points (:) du dernier exemple signale que le terme qui suit (mois) est soit un paramètre, soit une variable hôte.

Les expressions de chaînes

Les *expressions de chaînes* peuvent inclure l'opérateur de concaténation (‖). Utilisez la concaténation pour fusionner deux chaînes de caractères, comme le montre le Tableau 3.2.

TABLEAU 3.2 Exemples de concaténation de chaînes.

Expression	Résultat

'renseignement ' || 'militaire'	'renseignement militaire'
'tra' || 'duction'	'traduction'
VILLE|| '' ||ETAT|| ' '||CODE_POSTAL	Une seule chaîne composée de la ville, de l'État et du code postal, chacun étant séparé par un seul espace.

Certaines implémentations de SQL utilisent l'opérateur + pour la concaténation. Vérifiez ce point dans votre documentation.

Quelques implémentations peuvent proposer d'autres opérateurs de chaîne que la concaténation, mais le standard SQL ne les supporte pas.

Les expressions de valeurs de type date/heure et intervalle

Les *expressions de valeurs date/heure* manipulent des dates et des heures. Des données de type `DATE`, `TIME`, `TIMESTAMP` et `INTERVAL` peuvent figurer dans ces expressions. Le résultat d'une expression date/heure est une autre expression date/heure. Vous pouvez ajouter ou soustraire un intervalle d'une date/heure et spécifier des informations relatives au décalage horaire.

Voici un exemple d'utilisation d'une telle expression :

```
DATE_PRET + INTERVAL '7' DAY
```

Une bibliothèque pourrait utiliser une expression de ce type pour déterminer quand envoyer une note de rappel à un abonné pour qu'il rapporte les livres empruntés. L'exemple qui suit utilise cette fois-ci l'heure :

```
TIME '18:55:48' AT LOCAL
```

Le mot clé `AT LOCAL` signifie que l'heure est exprimée en temps national.

Les *expressions de type intervalle* évaluent le temps passé entre deux date/heure. Il y a deux types d'intervalles : *année-mois* et *jour-heure*. Vous ne pouvez pas les mélanger au sein d'une même expression.

Par exemple, quelqu'un pourrait rendre un livre à la bibliothèque après la date limite. En utilisant une expression de type intervalle, vous pourriez calculer combien de jours se sont écoulés depuis cette date butoir et calculer l'amende correspondante :

```
(DATE_RETOUR-DATE_LIMITE) DAY
```

Comme un intervalle peut être soit de type année-mois, soit de type jour-heure, vous devez préciser quel type vous souhaitez utiliser. Dans l'exemple précédent, j'ai spécifié *DAY*.

Les expressions de valeurs booléennes

Une expression de valeur booléenne teste la véracité d'un prédicat. Voici un exemple d'expression booléenne :

```
(CLASSE = DOCTORANT) IS TRUE
```

S'il s'agissait d'une condition pour récupérer des lignes d'une table qui recense des étudiants, seules celles qui correspondent à des doctorants vous seraient retournées. Pour obtenir les enregistrements de tous ceux qui ne sont pas doctorants, vous pourriez utiliser :

```
NOT (CLASSE = DOCTORANT) IS TRUE
```

Ce qui peut aussi s'écrire :

```
(CLASSE = DOCTORANT) IS FALSE
```

Voici comment récupérer toutes les lignes dont la colonne `CLASSE` contient une valeur nulle :

```
(CLASSE = DOCTORANT) IS UNKNOWN
```

Les expressions de valeurs de type défini

par l'utilisateur

Les types définis par l'utilisateur sont décrits dans le Chapitre 2. Cette fonctionnalité permet aux utilisateurs de définir leurs propres types de données au lieu d'utiliser ceux de SQL. Les expressions qui utilisent des données de type défini par l'utilisateur doivent évaluer un élément du même type.

Les expressions de valeurs de ligne

Une *expression valeur de ligne* spécifie, et ce n'est pas une surprise, une valeur de ligne. Celle-ci peut consister en une ou plusieurs expressions séparées par une virgule. Par exemple :

```
('Joseph Tykociner', 'Professeur Emérite', 1918)
```

Il s'agit d'une ligne d'une table décrivant toute l'histoire d'une université. Elle contient le nom, le rang et l'année d'intégration d'un professeur.

Les expressions de valeurs de collection

Une *expression valeur de collection* est évaluée comme un tableau.

Les expressions de valeurs de référence

Une *expression valeur de référence* est évaluée comme une valeur qui référence un autre composant de la base de données, comme une colonne de table.

Les prédicats

Les prédicats sont les équivalents SQL des propositions logiques. L'instruction suivante est un exemple de proposition :

```
«L'étudiant est un senior.»
```

Dans une table qui contient des informations sur les étudiants, le domaine de la colonne CLASSE peut être `SENIOR`, `JUNIOR`, `DIPLOME`, `THESARD`

ou NULL. Vous pouvez utiliser le prédicat CLASSE = SENIOR pour isoler toutes les lignes pour lesquelles le prédicat est faux ou toutes les lignes pour lesquelles le prédicat est vrai. Parfois, la valeur du prédicat pour une ligne est indéterminée (NULL). Dans ce cas, vous pourriez décider soit d'éliminer la ligne, soit de la retenir (après tout, l'étudiant pourrait être un senior). Le traitement adapté dépend du cas de figure.

CLASSE = SENIOR est un exemple de prédicat de comparaison. SQL dispose de six opérateurs de comparaison. Un prédicat de comparaison simple n'utilisera qu'un seul d'entre eux. Le Tableau 3.3 contient une liste de prédicats de comparaison et des exemples de leur utilisation.

Dans l'exemple précédent, seules les deux premières entrées (CLASSE = SENIOR et CLASSE <> SENIOR) ont un sens, les autres comparaisons n'ayant d'autre signification que celle d'un tri alphabétique, ce qui n'est peut-être pas ce que vous attendiez.

TABLEAU 3.3 Opérateurs de comparaison et prédicats de comparaison.

Opérateur	Comparaison	Expression
=	Egal à	CLASSE = SENIOR
<>	Différent de	CLASSE <> SENIOR
<	Inférieur à	CLASSE < SENIOR
>	Supérieur à	CLASSE > SENIOR
<=	Inférieur ou égal à	CLASSE <= SENIOR
>=	Supérieur ou égal à	CLASSE >= SENIOR

Connecteurs logiques

Les connecteurs logiques vous permettent de construire des prédicats complexes à partir d'expressions plus simples. Supposons par exemple que vous souhaitiez identifier les enfants prodiges dans une base de données d'étudiants. Deux propositions pourraient identifier ces enfants :

```
«L’étudiant est un senior»
«L’âge de l’étudiant est inférieur à 14»
```

Vous pouvez faire appel au connecteur logique AND pour fabriquer un prédicat composé qui isole les enregistrements correspondant aux étudiants recherchés. Par exemple :

```
CLASSE = SENIOR AND AGE < 14
```

Si vous utilisez le connecteur `AND`, les deux prédicats doivent être validés (vrais) pour que le prédicat composé le soit aussi. Utilisez le connecteur `OR` quand vous voulez que le prédicat composé soit validé si l’un ou l’autre des prédicats qui le compose est validé (ou les deux). `NOT` est le troisième connecteur logique. À proprement parler, `NOT` ne permet pas de connecter des prédicats, mais il inverse la valeur de l’expression à laquelle vous l’appliquez. Prenons un exemple :

```
NOT (CLASSE = SENIOR)
```

Cette expression est validée si `CLASSE` n’est pas égal à `SENIOR`.

Les fonctions d’ensemble

Quelquefois, l’information que vous voulez extraire d’une table ne dépend pas du contenu d’une ligne, mais de plusieurs lignes. SQL dispose de cinq fonctions d’ensemble (ou) pour ce cas. Ces fonctions sont `COUNT`, `MAX`, `MIN`, SUM et `AVG`. Chaque fonction extrait des données d’un ensemble de lignes.

COUNT

La fonction `COUNT` retourne le nombre de lignes de la table spécifiée qui vérifient le critère spécifié. Pour compter le nombre de « seniors » précoces dans la base de données d’étudiants, utilisez par exemple l’instruction suivante :

```
SELECT COUNT (*)
```

```
FROM ETUDIANT
WHERE GRADE = 12 AND AGE < 14 ;
```

MAX

La fonction `MAX` renvoie la valeur maximale que prend une colonne dans un ensemble de lignes. Supposons que vous souhaitiez trouver l'étudiant qui a le plus d'ancienneté dans votre école. Vous utiliseriez une instruction comme celle-ci :

```
SELECT NOM, PRENOM, AGE
FROM ETUDIANT
WHERE AGE = (SELECT MAX(AGE) FROM ETUDIANT);
```

Cette instruction retourne tous les étudiants dont l'âge est égal à l'âge maximal. C'est-à-dire que si l'âge de l'étudiant le plus ancien est 23, cette instruction retournera les noms et les prénoms de tous les étudiants qui ont 23 ans.

Cette requête utilise une sous-requête. La sous-requête `SELECT MAX(AGE) FROM ETUDIANT` est incluse dans la requête principale.

MIN

La fonction `MIN` fonctionne exactement comme la fonction `MAX`, à ceci près qu'elle recherche la valeur minimale d'une colonne dans un ensemble de lignes. Pour trouver le plus jeune étudiant, vous pourriez utiliser la requête suivante :

```
SELECT NOM, PRENOM, AGE
FROM ETUDIANT
WHERE AGE = (SELECT MIN(AGE) FROM ETUDIANT);
```

Cette requête retourne tous les étudiants dont l'âge est celui de l'étudiant le plus jeune.

SUM

La fonction SUM effectue la somme des valeurs d'une colonne donnée. La colonne doit posséder l'un des types de données numériques reconnus, et la valeur de la somme doit appartenir à la plage des valeurs autorisées pour cette colonne. Par exemple, si la colonne est de type SMALLINT, la somme ne doit pas dépasser la limite supérieure du type de données SMALLINT. Dans la base de données des clients que nous avons utilisée plus haut, la table FACTURE contient un enregistrement de toutes les ventes. Pour trouver la valeur totale de celles-ci, utilisez la fonction SUM de la manière suivante :

```
SELECT SUM(TOTAL_VENTE) FROM FACTURE;
```

AVG

La fonction AVG retourne la moyenne de toutes les valeurs d'une colonne donnée. Comme la fonction SUM, AVG ne s'applique qu'aux colonnes dont le type de données est numérique. Pour trouver la valeur moyenne des ventes, appelez la fonction AVG de la manière suivante :

```
SELECT AVG(TOTAL_VENTE) FROM FACTURE
```

Souvenez-vous que les valeurs nulles n'ont pas de valeurs. Par conséquent, si des lignes de TOTAL_VENTE contiennent des valeurs nulles (inconnues ou non encore définies), elles seront purement et simplement ignorées dans le calcul de la moyenne.

Sous-requêtes

Comme vous avez pu le constater dans la section précédente, les sous-requêtes sont des requêtes insérées dans d'autres requêtes. Partout où vous pouvez utiliser une expression, vous pouvez aussi utiliser une sous-requête. Les sous-requêtes sont particulièrement utiles pour relier l'information que contient une table à celle qui se trouve dans une autre table. Vous pouvez en effet inclure une requête portant sur une table dans une requête concernant une autre table. En imbriquant plusieurs sous-requêtes, il est possible d'accéder aux informations provenant de deux ou de plusieurs tables pour générer le résultat final.

Le langage de contrôle de données (DCL)

Le langage de contrôle de données (DCL) dispose de quatre instructions : `COMMIT`, `ROLLBACK`, `GRANT` et `REVOKE`. Ces instructions servent toutes à protéger la base de données de modifications accidentelles ou intentionnelles qui pourraient mettre en danger son intégrité.

Les transactions

Votre base de données est plus vulnérable aux dangers pendant que vous ou quiconque la modifiez. Même s'il n'y a qu'un seul utilisateur, toute modification peut se révéler dangereuse pour la base de données. Une panne logicielle ou matérielle alors que l'édition est en cours peut laisser la base de données dans un état instable entre ce qu'elle était avant la modification et ce qu'elle aurait dû être après cette modification.

SQL protège votre base de données en limitant les opérations susceptibles d'affecter la base de données de telle sorte que ces opérations ne peuvent se dérouler que dans le cadre de transactions. Durant une transaction, SQL enregistre chaque opération sur les données dans un fichier journal (ou log). Si un événement quelconque interrompt la transaction avant que l'instruction `COMMIT` n'y mette un terme, vous pouvez restaurer le système dans son état initial en utilisant l'instruction `ROLLBACK`. `ROLLBACK` exécute le fichier de log dans le sens inverse, annulant toutes les actions qui y figurent. Une fois restauré l'état initial de la base de données, vous pouvez rechercher la cause du problème et procéder à une nouvelle transaction.

Tant qu'un problème logiciel ou matériel peut survenir, votre base de données est susceptible d'être endommagée. Pour minimiser les risques, les SGBD effectuent toutes les opérations qui modifient la base de données dans le contexte de transactions et valident ces opérations les unes après les autres. Les systèmes de gestion de bases de données modernes utilisent le logging en plus des transactions pour garantir qu'un problème logiciel, matériel ou opérationnel ne causera pas de dommages à la base de données. Une fois que la transaction a été validée, elle est préservée de tout (sauf des catastrophes majeures !). Insistons encore : tant que l'instruction `COMMIT` n'a pas été exécutée, la machine à remonter le temps permet de revenir

jusqu'au point de départ d'une transaction. Une fois le problème corrigé, l'histoire peut reprendre son fil normal.

Sur un système multi-utilisateur, une base de données risque d'être corrompue et/ou de produire des résultats erronés sans que survienne une quelconque panne logicielle ou matérielle. Les interactions entre les différents utilisateurs qui accèdent à la même table au même instant sont en effet susceptibles de poser de sérieux problèmes. En limitant les modifications de sorte qu'elles ne puissent se produire que dans le cadre de transactions, SQL résout aussi cette difficulté.

En effectuant toutes les opérations qui peuvent affecter la base de données dans le cadre de transactions, vous pouvez isoler les actions d'un utilisateur de celles d'un autre utilisateur. Cela est vital pour garantir que les résultats seront corrects.

Vous vous demandez peut-être comment les interactions entre deux utilisateurs peuvent produire des résultats erronés. Supposons par exemple que Didier lise un enregistrement dans une table d'une base de données. Un instant plus tard, David modifie la valeur d'un champ numérique dans cet enregistrement. Didier écrit alors au même emplacement une donnée qui est fonction de ce qu'il a lu quelques secondes plus tôt. Comme Didier n'est pas au courant des modifications de David, la valeur qu'il saisit n'est pas correcte.

Un autre problème peut survenir si Didier écrit dans un enregistrement puis que David lit ce dernier. Si Didier annule sa transaction, David ne sera pas informé de cette opération, et il procédera à des modifications sur la base d'informations qui ne sont plus à jour.

Les utilisateurs et les privilèges

Hormis la corruption des données qui résulte de pannes logicielles ou matérielles, ou encore d'interactions malencontreuses entre utilisateurs, un autre danger menace l'intégrité d'une base de données : les utilisateurs eux-mêmes. Certaines personnes ne devraient pas pouvoir accéder du tout aux données. D'autres devraient ne pouvoir accéder qu'à certaines données, mais pas à toutes. Et d'autres encore devraient pouvoir accéder à tout sans aucune contrainte. Vous avez donc besoin d'un système permettant de cataloguer les utilisateurs et d'affecter (ou de refuser) des privilèges aux membres des différentes catégories.

Le créateur d'un schéma spécifie qui est considéré comme étant son propriétaire. En tant que propriétaire du schéma, vous avez le droit d'accorder des privilèges d'accès à vos utilisateurs. Tout privilège que vous n'accordez pas explicitement est considéré comme étant refusé. Les privilèges sont aussi susceptibles d'être retirés à tout moment. Un utilisateur doit passer par une procédure d'authentification afin de prouver son identité. Ce n'est qu'après avoir montré patte blanche qu'il sera capable d'accéder aux fichiers et aux données pour lesquels il dispose de privilèges. La procédure exacte dépend de l'implémentation (voire du matériel si la sécurité exige une lecture d'empreintes digitales ou de l'iris).

SQL vous permet de protéger les objets suivants :

- Les tables.
- Les colonnes.
- Les vues.
- Les domaines.
- Les jeux de caractères.
- Les séquences d'interclassement.
- Les traductions.

Je traite des jeux de caractères, des séquences d'interclassement et des traductions dans le Chapitre 5.

SQL supporte plusieurs types de protection : voir, ajouter, modifier, supprimer, référencer et utiliser les bases de données. Il existe de surcroît des protections associées à l'exécution de routines externes.

Vous accordez un privilège d'accès en faisant appel à l'instruction `GRANT` et vous le supprimez avec l'instruction `REVOKE`. En supervisant le fonctionnement de l'instruction `SELECT`, le DCL contrôle qui a le droit de voir un objet de la base de données comme une table, une colonne ou encore une vue. De même, la maîtrise sur l'instruction `INSERT` permet de déterminer qui peut ajouter de nouvelles lignes à une table. De même aussi, restreindre l'emploi de l'instruction `UPDATE` permet de n'autoriser la modification des lignes des tables qu'aux seuls utilisateurs autorisés. Et

restreindre l'usage de l'instruction `DELETE` sert à définir qui a le droit de supprimer des lignes dans des tables.

Si une table d'une base de données définit comme clé étrangère une colonne qui est clé primaire dans une autre table, vous pouvez ajouter une contrainte à la première table de sorte qu'elle référence la seconde (les clés étrangères sont décrites dans le Chapitre 5). Quand une table en référence ainsi une autre, son propriétaire peut être à même de déduire des informations sur le contenu de la seconde. Et si vous êtes le propriétaire de la seconde table, il n'est pas certain que cette perspective vous enchante. L'instruction `GRANT REFERENCES` de SQL vous donne le pouvoir d'empêcher une telle intrusion. La section suivante présente le problème posé par une référence inattendue et montre comment l'instruction `GRANT REFERENCES` permet de le résoudre. Grâce à l'instruction `GRANT USAGE`, il est possible de contrôler qui a le droit d'utiliser ou de visualiser le contenu d'un domaine, d'un jeu de caractères, d'une séquence d'interclassement ou d'une traduction.

Le Tableau 3.4 contient la liste des instructions que SQL met à votre disposition pour accorder et retirer des privilèges.

TABLEAU 3.4 Les types de protection.

Opération de protection	Instruction
Permettre de voir une table	`GRANT SELECT`
Interdire de voir une table	`REVOKE SELECT`
Permettre d'ajouter des lignes à une table	`GRANT INSERT`
Interdire d'ajouter des lignes à une table	`REVOKE INSERT`
Permettre de modifier des données dans des lignes d'une table	`GRANT UPDATE`

Interdire de modifier des données dans des lignes d'une table	`REVOKE UPDATE`
Permettre de supprimer des lignes dans une table	`GRANT DELETE`
Interdire de supprimer des lignes dans une table	`REVOKE DELETE`
Permettre de référencer une table	`GRANT REFERENCES`
Interdire de référencer une table	`REVOKE REFERENCES`
Permettre d'utiliser un domaine, une traduction de caractères ou une collation	`GRANT USAGE ON DOMAIN, GRANT USAGE ON CHARACTER SET, GRANT USAGE ON TRANSLATION`
Interdire d'utiliser un domaine, une traduction de caractères ou une collation	`REVOKE USAGE ON DOMAIN, REVOKE USAGE ON CHARACTER SET, REVOKE USAGE ON TRANSLATION`

Vous pouvez attribuer différents niveaux d'accès à différentes personnes, et ce en fonction de leurs besoins. Voici quelques exemples :

```
GRANT SELECT
ON CLIENT
TO RESPONSABLE_VENTES;
```

permet à une personne, en l'occurrence le responsable des ventes, de voir la table CLIENT.

L'exemple suivant autorise quiconque disposant d'un accès au système de voir la liste des prix publics :

```
GRANT SELECT
ON PRIX_PUBLICS
TO PUBLIC;
```

Nous voulons maintenant permettre au responsable des ventes d'éditer la liste des prix publics. Il doit avoir la possibilité de modifier le contenu de lignes déjà présentes, mais sans pouvoir ni en supprimer ni en ajouter :

```
GRANT UPDATE
ON PRIX_PUBLICS
TO RESPONSABLE_VENTES;
```

Si le responsable des ventes est autorisé à ajouter de nouvelles lignes dans la liste des prix publics, nous écrirons :

```
GRANT INSERT
ON PRIX_PUBLICS
TO RESPONSABLE_VENTES;
```

Grâce à ce dernier exemple, le rôle du responsable des ventes s'accroît encore, et il peut maintenant supprimer des lignes de la table :

```
GRANT DELETE
ON PRIX_PUBLICS
TO RESPONSABLE_VENTES;
```

Les contraintes d'intégrité référentielle peuvent compromettre vos données

Vous pourriez croire que le contrôle des fonctions de visualisation, de création, de modification et de suppression dans une table suffit à vous protéger. En fait, un pirate compétent pourrait très bien vous dévaliser en utilisant une autre méthode.

Une base de données relationnelle correctement conçue possède une intégrité référentielle, ce qui signifie que les données d'une table de cette base sont en cohérence avec les données de toutes les autres tables. Pour assurer cette intégrité référentielle, les concepteurs de bases de données appliquent aux tables des contraintes qui limitent ce qu'il est possible d'y saisir. Si votre base de données est dotée de contraintes d'intégrité référentielle, un utilisateur peut avoir la possibilité de créer une nouvelle table utilisant comme clé étrangère une colonne d'une table confidentielle. Cette colonne sert alors de lien via lequel quelqu'un peut subtiliser des informations confidentielles.

Précisons cette idée. Vous êtes un célèbre analyste boursier, disons de Wall Street. Beaucoup de gens croient en la pertinence de vos achats, si bien que chaque fois que vous négociez des actions pour le compte de vos clients, de nombreuses personnes suivent le mouvement, ce qui fait monter les cours. Vous conservez vos analyses dans une base de données qui contient une table nommée `QUATRE_ETOILES`. Toutes vos recommandations les plus précieuses y sont stockées. Bien entendu, vous restreignez l'accès à la table `QUATRE_ETOILES` de manière 1) que seuls vos clients payants puissent y accéder et 2) qu'ils ne puissent le faire que quand vous les y autorisez (par le biais par exemple d'une lettre d'information périodique).

Cependant, vous restez vulnérable si quelqu'un peut créer une nouvelle table qui utilise le nom d'une colonne de `QUATRE_ ETOILES` comme clé étrangère, comme dans l'exemple suivant :

```
CREATE TABLE BONNES_ACTIONS (
ACTION CHARACTER (30) REFERENCES QUATRE_ETOILES
);
```

Le pirate peut maintenant essayer d'insérer le nom de chaque action cotée en Bourse. Les insertions réussies lui indiqueront quelles actions sont stockées dans votre table confidentielle, et il ne lui faudra pas beaucoup de temps pour piller vos si précieuses informations.

Vous pouvez vous protéger de tels assauts en faisant très attention lorsque vous saisissez des instructions telles que celle-ci :

```
GRANT REFERENCES (ACTION)
ON QUATRE_ETOILES
```

```
TO JESUISUN_HACKER;
```

Clairement, j'exagère ici. Vous ne donneriez jamais accès à une table critique à une personne qui ne serait pas digne de confiance, n'est-ce pas ? Du moins, si vous avez réalisé ce que vous faites. Toutefois, les pirates ne maîtrisent pas seulement la technique de nos jours. Ils sont aussi des experts de l'*ingénierie sociale*, l'art de tromper les gens pour leur faire faire ce qu'ils ne feraient pas normalement. Méfiez-vous de ceux qui mentionnent quoi que ce soit en lien avec des informations confidentielles.

Évitez d'accorder des privilèges à des personnes qui pourraient en abuser. Vous n'êtes pas disposé à prêter votre voiture à quelqu'un pour un long voyage ? Alors ne lui accordez pas de privilège `REFERENCE` sur une table que vous jugez importante.

La liste suivante décrit deux autres bonnes raisons de n'accorder des privilèges qu'avec parcimonie :

- Si l'autre personne (le pirate) spécifie une contrainte sur `BONNES_ACTIONS` en utilisant l'option `RESTRICT` et que vous tentez ensuite de supprimer une ligne de votre table, le SGBD vous informera que vous ne pouvez pas le faire car ce serait violer l'intégrité relationnelle.
- Si vous voulez utiliser la commande `DROP` pour détruire votre table, vous verrez qu'il vous faudra d'abord attendre que l'autre personne (toujours le méchant pirate) ait supprimé sa contrainte (ou sa table).

Pour conclure, retenez que permettre à une autre personne de spécifier des contraintes d'intégrité sur votre table génère non seulement des failles de sécurité, mais aussi la met dans une position où elle pourrait vous barrer le chemin…

Déléguer la responsabilité de la sécurité

Si vous voulez que votre système reste sécurisé, vous devez sévèrement limiter l'attribution des privilèges d'accès ainsi que le nombre de personnes auxquelles vous les accordez. Cependant, les gens qui n'arrivent pas à travailler parce qu'ils sont bloqués par de trop grandes restrictions risquent de vous harceler du matin au soir. Pour éviter la dépression, n'hésitez pas à déléguer certaines responsabilités dans la gestion de la sécurité de votre base de données. Considérez l'exemple suivant :

```
GRANT UPDATE
ON PRIX_PUBLICS
TO RESPONSABLE_VENTES WITH GRANT OPTION
```

Cette instruction ressemble au précédent exemple `GRANT UPDATE` en ce que cette instruction permet au responsable des ventes de mettre à jour la liste des prix. Cependant, elle lui donne aussi le droit de rétrocéder ce privilège à n'importe quelle autre personne. Si vous utilisez cette forme de l'instruction GRANT, vous choisissez non seulement de faire confiance à la personne à laquelle vous accordez le privilège, mais aussi à celles à qui cette personne le transmettra.

La forme ultime de la confiance, et donc la forme ultime de vulnérabilité, est d'exécuter une instruction telle que celle-ci :

```
GRANT ALL PRIVILEGES
ON QUATRE_ETOILES
TO BENEDICT_ARNOLD WITH GRANT OPTION;
```

Faites très attention quand vous utilisez de telles instructions (puisque vous donnez par ricochet tous les privilèges à la planète entière).

Utiliser SQL pour créer des bases de données

DANS CETTE PARTIE :

Créer des structures simples.

Établir des relations entre les tables.

Chapitre 4
Créer et maintenir une simple structure de base de données

DANS CE CHAPITRE :

- **Créer, modifier et supprimer une table de base de données en utilisant un RAD.**
- **Créer, modifier et supprimer une table de base de données en utilisant SQL.**
- **Migrer votre base de données vers un autre SGBD.**

L'informatique évolue si vite que la succession des « générations » peut parfois paraître confuse. Les langages de haut niveau (ou de troisième génération) tels que FORTRAN, COBOL, Basic, Pascal et C furent les premiers à utiliser les bases de données. Quelque temps plus tard, des langages spécifiquement conçus pour les bases de données tels que dBase, Paradox et R:BASE (troisième génération et demie ?) firent leur apparition. La dernière étape de cette évolution est l'émergence d'environnements de développement tels qu'Access, Delphi, IntraBuilder et C++ Builder (langages de quatrième génération ou L4G) qui permettent de créer des applications avec très peu de programmation procédurale, voire aucune. Mais nous n'en sommes déjà plus à ces générations, depuis l'apparition d'environnements de développement rapides (RAD, pour Rapid Application Development) et les environnements intégrés de développement (IDE pour Integrated Development Environnments) tels qu'Eclipse et Visual Studio. NET, qui peuvent être utilisés avec de nombreux langages (tels que C, C++, C#, Python, Java, Visual Basic ou PHP). Vous pouvez les utiliser pour assembler des composants applicatifs et réaliser ainsi une application.

Du fait que SQL n'est pas un langage complet, il n'appartient pas à une des catégories que je viens de présenter. Il utilise des instructions à la manière des langages de troisième génération, mais il est pour l'essentiel non

procédural, comme le sont les L4G. Mais peu importe la classification de SQL. Vous pouvez l'utiliser en conjonction avec tous les principaux outils de développement des langages de troisième et quatrième génération. Vous pouvez écrire le code SQL vous-même ou le faire générer par l'environnement de développement. Le code produit est dans les deux cas du pur SQL.

Dans ce chapitre, je vais vous expliquer comment créer, modifier et supprimer une table en utilisant un RAD. Je vous montrerai ensuite comment faire la même chose en SQL.

Créer une simple base de données en utilisant un outil RAD

Les gens utilisent des bases de données pour conserver la trace d'informations importantes. Parfois l'information qu'ils cherchent à enregistrer est simple, parfois elle ne l'est pas. Un bon système de gestion de bases de données fournit ce dont vous avez besoin dans chaque cas. Quelques-uns s'appuient sur SQL. D'autres, comme les outils RAD, vous proposent un environnement graphique orienté objet. Quelques SGBD proposent les deux. Dans les sections suivantes, je vous montrerai comment créer une base de données à l'aide d'un outil graphique. J'utilise Microsoft Access, mais la procédure est la même avec d'autres environnements de développement pour Windows.

Un scénario probable

La première étape dans la création d'une base de données consiste à déterminer les informations que vous voulez conserver. Prenons un exemple : supposons que vous veniez de gagner 11 millions d'euros au Loto. Des gens dont vous n'aviez plus entendu parler depuis des années et des amis de trente ans se rappellent à votre bon souvenir. Certains vous proposent d'investir dans des affaires qui rapporteront à coup sûr. D'autres représentent des causes pour lesquelles vous devriez faire des dons. En bon gestionnaire, vous avez tôt fait de réaliser que certaines affaires semblent moins intéressantes que d'autres. Et que certaines causes valent moins qu'on les défende que d'autres. Vous décidez de stocker toutes les

propositions qui vous sont faites dans une base de données afin de les recenser et de les étudier équitablement.

Vous choisissez de conserver la trace des informations suivantes :

- Nom.
- Prénom.
- Adresse.
- Ville.
- État ou province.
- Code postal.
- Téléphone.
- Relation (votre relation à cette personne).
- Proposition.
- Affaire ou œuvre.

Comme vous ne voulez pas vous perdre en subtilités, vous décidez de stocker toutes ces informations dans une seule table de la base de données.

Créer une table de base de données

Lorsque vous lancez l'environnement de développement Access 2013, vous êtes accueilli par l'écran repris sur la Figure 4.1. De là, vous pouvez créer une table de base de données de différentes manières. Je vais commencer par le mode Feuille de données, car elle vous montre comment créer une base de données à partir de rien.

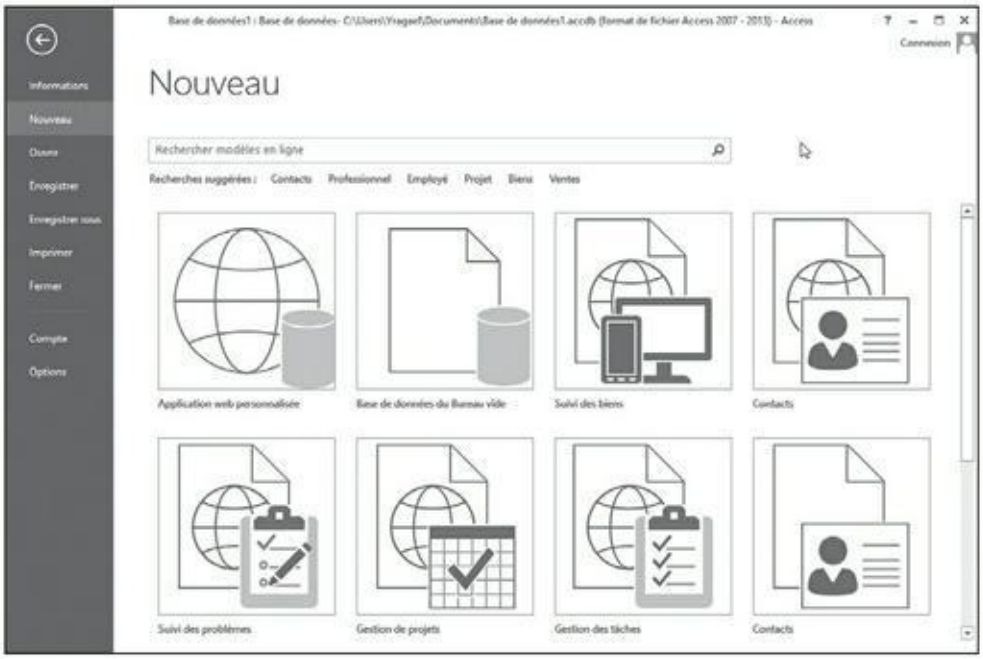


FIGURE 4.1 L'écran d'accueil de Microsoft Access.

Créer une table de base de données dans le mode Feuille de données

Par défaut, Access 2013 s'ouvre sur le mode Feuille de données. Pour créer une base de données Access dans cette vue, double-cliquez sur le modèle Base de données du Bureau vide. Votre feuille de données Access attend que vous saisissiez des données dans Table 1, la première table de votre base de données, comme sur la Figure 4.2. Vous pourrez modifier le nom de la table pour lui donner un nom plus signifiant ultérieurement. Access donne à votre base de données le nom par défaut `Base de données1` (ou `Base de données31` si vous avez déjà créé 30 bases de données et que vous ne vous êtes pas donné la peine de les renommer). Il vaut mieux donner un nom signifiant à la base de données pour éviter toute confusion.

C'est une méthode qui part de rien, mais il existe d'autres manières de créer une table dans une base de données Access. Par la suite, nous utiliserons le mode Création.

Créer une table dans une base de données en vue Création

Dans le mode Feuille de données (représentée sur la Figure 4.2), il est très facile de créer une table de base de données : vous n'avez qu'à commencer à saisir les données. Toutefois, cette manière de faire est sujette aux erreurs, car on oublie rapidement des détails. Il vaut mieux utiliser le mode Création en suivant les étapes suivantes :

FIGURE 4.2 Le mode Feuille de données dans l'environnement de développement Access.

1. **Dans le mode Feuille de calcul (par défaut), cliquez sur l'onglet Accueil dans le ruban et cliquez sur Affichage en dessous de l'icône qui se trouve dans l'angle supérieur gauche de la fenêtre. Sélectionnez Mode création dans le menu déroulant.**

   Lorsque vous choisissez le mode Création, une boîte de dialogue apparaît pour vous demander le nom de la table.

2. **Saisissez LOTO et cliquez OK.**

Le mode Création apparaît (représenté sur la Figure 4.3).

Notez que la fenêtre est divisée en zones fonctionnelles. Deux d'entre elles sont particulièrement utiles pour créer des tables de bases de données :

- *Les options du mode Création* : Un menu en haut de la fenêtre donne accès aux options Accueil, Créer, Données externes, Outils de base de données et Création.

  Lorsque ce ruban est affiché, les outils disponibles dans le mode Création sont représentés par des icônes se trouvant immédiatement sous le menu. Sur la Figure 4.3, la mise en exergue montre que les icônes Création et Clé primaire sont sélectionnées.

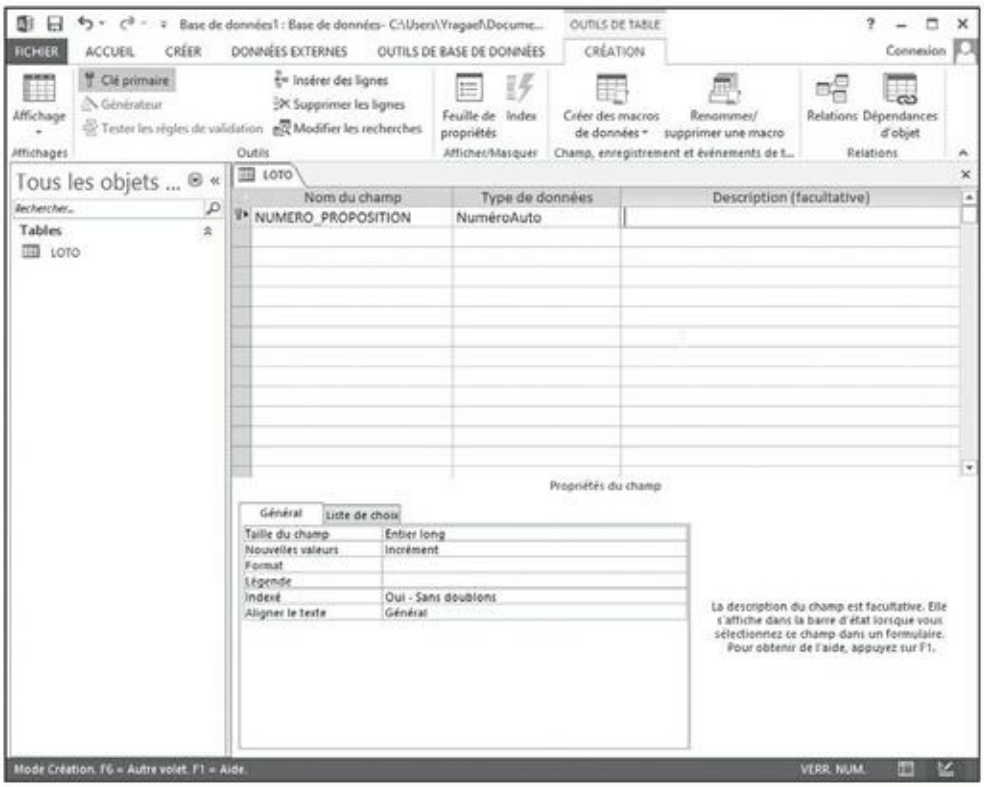


FIGURE 4.3 L'écran de démarrage du mode Création.

- *Le panneau des propriétés du champ* : Dans cette zone qui permet de définir les champs de la base de données, le curseur clignote dans la colonne Nom du champ de la première ligne. Access suggère que vous spécifiez ici une clé primaire, que vous la nommiez `ID`, et il lui donnera automatiquement le type de données `NuméroAuto`.

NuméroAuto est un type de données Access, et non un type de données standard en SQL. Il incrémente un entier dans le champ chaque fois que vous ajoutez un nouvel enregistrement dans la table. Ce type de données vous garantit que le champ que vous utilisez comme clé primaire ne prendra pas deux fois la même valeur, toute clé primaire devant être unique.

3. **Dans la zone Propriétés du champ, modifiez Nom du champ pour la clé primaire en le passant de `ID` à `NUMERO_PROPOSITION`.**

Le nom proposé dans Nom du champ pour la clé primaire, `ID`, n'est pas très explicite. Si vous prenez l'habitude de le changer en quelque chose de plus signifiant (et/ou de fournir une description dans la colonne Description), il sera plus facile de garder une trace de la raison d'être des champs dans votre base de données. Ici, les noms des champs sont suffisamment explicites.

La Figure 4.4 vous montre à quoi ressemble la création de la table de la base de données à cette étape.

4. **Dans le panneau Propriétés du champ, vérifiez les hypothèses formulées automatiquement par Access sur le champ** `NUMERO_PROPOSITION`**.**

   La Figure 4.4 montre que ces hypothèses sont les suivantes :

   - La taille du champ est Entier long.
   - Les nouvelles valeurs sont générées par incrémentation.

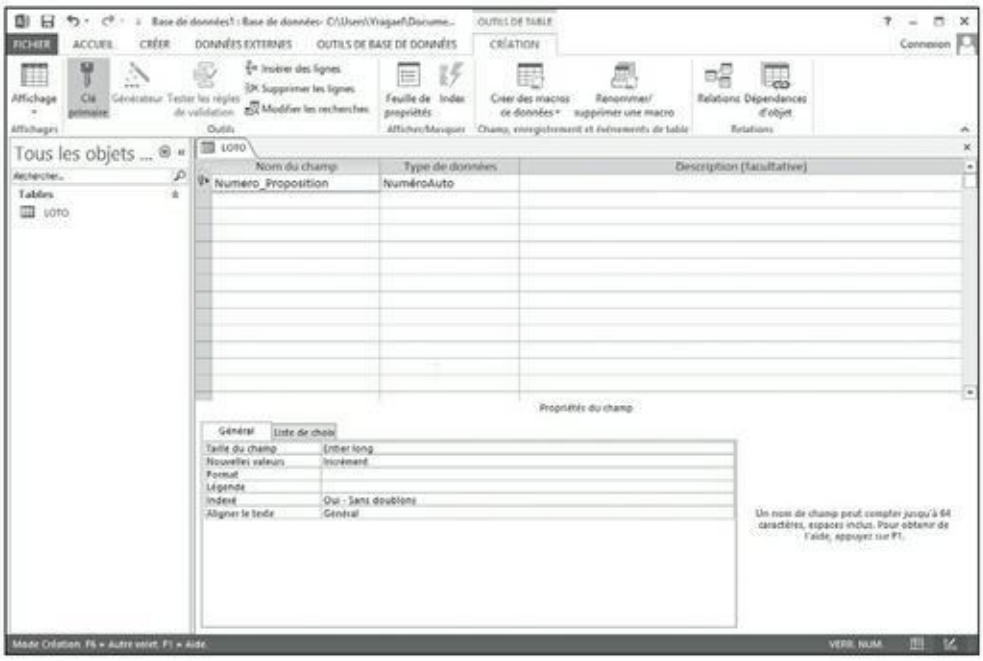


FIGURE 4.4 Utilisez un nom de champ signifiant pour définir la clé primaire.

   - L'indexation est activée et les doublons ne sont pas autorisés.
   - L'alignement du texte est général.

   Comme c'est souvent le cas, les hypothèses formulées par Access conviennent à ce que vous souhaitez faire. Si ce

n'était pas le cas, vous pourriez les modifier en saisissant de nouvelles valeurs.

5. **Spécifiez le reste des champs que vous souhaitez voir dans la table.**

   La Figure 4.5 vous montre le mode Création une fois que vous avez saisi le champ Prénom.

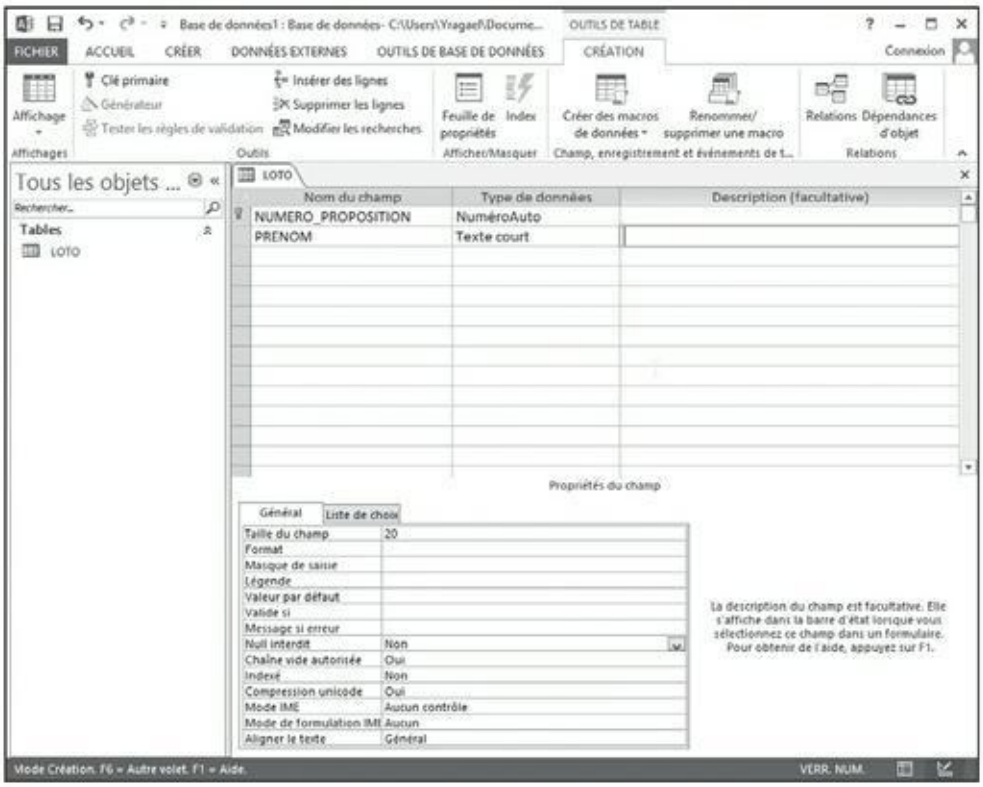


FIGURE 4.5 La fenêtre de création de la table une fois que vous avez défini Prénom.

Le type de données pour `PRENOM` est `Texte court` plutôt que `NuméroAuto`, si bien que les propriétés du champ sont différentes. En l'occurrence, Access a donné à `PRENOM` la taille de champ par défaut, c'est-à-dire 255 caractères. Je ne connais pas grand monde dont le prénom soit aussi long. Access est assez futé pour n'allouer de mémoire qu'en fonction de ce qui est saisi. Il n'alloue pas 255 octets à l'aveugle. Toutefois, d'autres environnements de

développement peuvent ne pas en être capables. J'aime préciser une valeur raisonnable comme taille de champ. Cela m'évite des problèmes quand je passe d'un environnement de développement à un autre.

L'hypothèse par défaut d'Access est que `Prénom` n'est pas un champ obligatoire. Vous pourriez saisir un enregistrement dans la table LOTO et laisser ce champ vide, ce qui permettrait de tenir compte des gens qui n'ont qu'un nom, comme Cher ou Bono.

ANTICIPEZ LORSQUE VOUS CONCEVEZ VOTRE TABLE

Dans certains environnements de développement (différents d'Access), réduire la taille du champ `Prénom` à 15 octets permet d'économiser 240 octets pour *chaque enregistrement dans la base de données* si vous utilisez des caractères ASCII (UTF-8), 480 octets si vous utilisez des caractères UTF-16, ou 960 octets si vous utilisez des caractères UTF-32. L'économie devient vite considérable. Tant que vous y êtes, jetez un œil sur les autres hypothèses par défaut formulées sur les autres propriétés du champ, et essayez d'anticiper comment vous voudrez les utiliser tandis que la base de données s'enrichira. Il faut se préoccuper de certaines tout de suite pour plus d'efficacité (la taille du champ est un bon exemple) ; d'autres ne s'appliqueront que dans des cas obscurs.

Vous aurez sans doute remarqué qu'une autre propriété de champ revient souvent : la propriété `Indexé`. Si vous n'envisagez pas de récupérer la valeur du champ, ne gaspillez pas la puissance de votre ordinateur pour l'indexer. Toutefois, notez que dans une grande table comportant de nombreuses lignes, vous pouvez accélérer

considérablement la récupération en indexant le champ que vous utilisez pour identifier l'enregistrement que vous souhaitez récupérer. En matière de conception de tables de bases de données, le diable se loge dans les détails.

6. **Passez la taille du champ à 15.**

   Pour savoir pourquoi c'est une bonne idée, reportez-vous à l'encadré « Anticipez lorsque vous concevez votre table ».

7. **Pour vous assurer que vous pouvez récupérer un enregistrement rapidement dans la table LOTO à partir de `Prénom`, passez la propriété `Indexé` de ce champ à `Oui`, comme sur la Figure 4.6.**

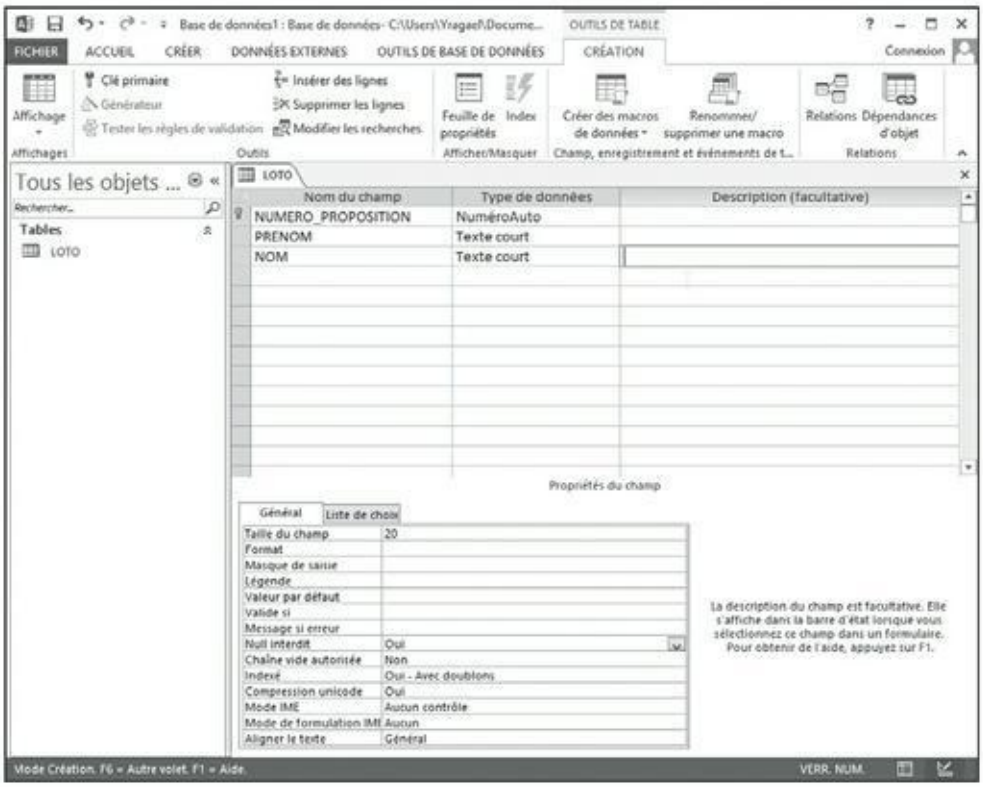


FIGURE 4.6 La fenêtre de création une fois que Prénom a été défini.

La Figure montre quelques modifications que j'ai apportées dans le panneau Propriétés du champ :

- J'ai réduit la taille maximale du champ de 255 à 20.
- J'ai passé Null interdit à Oui, Chaîne vide autorisée à Non et Indexé à Oui - Avec doublons. Je veux que chaque proposition comprenne le nom de famille de la personne qui en est l'auteur. Un nom de taille nulle n'est pas autorisé, et le champ `Nom` doit être indexé.
- J'ai autorisé les doublons ; deux ou plusieurs personnes peuvent parfaitement avoir le même nom de famille. C'est pratiquement certain dans le cas de la table LOTO ; j'attends des propositions de mes trois frères, ainsi que de mes enfants et de ma fille qui n'est pas encore mariée, sans mentionner mes cousins.
- L'option Oui - Sans doublons, que je n'ai pas choisie, serait adaptée pour un champ qui serait la clé primaire de la table. En effet, la clé primaire d'un enregistrement devrait toujours être unique.

8. **Saisissez le reste des champs, en modifiant la propriété Taille du champ comme nécessaire selon le cas.**

   La Figure 4.7 vous présente le résultat.

FIGURE 4.7 La fenêtre de création de la table une fois que tous les champs ont été définis.

Comme vous pouvez le constater sur la Figure 4.7, le champ pour les entreprises et les associations (`AFFAIRE_ OEUVRE`) n'est pas indexé. Il n'est pas utile d'indexer un champ qui n'a que deux entrées possibles ; l'indexation ne viendrait pas réduire la sélection dans une proportion telle qu'elle en vaudrait la peine.

Access utilise le terme de *champ* plutôt que celui d'*attribut* ou de *colonne*. Le programme n'était pas initialement relationnel ; il utilisait une terminologie à base de *fichier*, *champ* et *enregistrement*, commune aux systèmes à base de fichiers à plat.

9. **Enregistrez votre table en cliquant sur l'icône figurant une disquette dans l'angle supérieur gauche de la fenêtre.**

Il est bon d'anticiper lorsque vous développez une base de données. Par exemple, c'est une bonne idée que d'enregistrer fréquemment votre travail, cliquez simplement sur l'icône figurant une disquette de temps en temps. De la sorte, vous vous épargnerez de reprendre votre travail s'il survient une coupure de courant ou un évènement imprévu. Aussi, même si vous pouvez parfaitement donner le même nom à la base de données et à l'une de ses tables, vous risquez d'induire en confusion les administrateurs et les utilisateurs par la suite. Il vaut donc mieux se faire une règle d'utiliser des noms différents.

Une fois que vous aurez enregistré votre table, vous découvrirez que vous devrez modifier un peu votre création, comme je l'explique dans la section suivante « Modifier la structure de votre table ».

Modifier la structure de votre table

Bien souvent, les bases de données que vous concevez ne tournent pas rond dès la première fois. Si vous travaillez pour le compte d'un client, vous pouvez être quasiment certain que ce dernier va revenir à la charge pour vous demander de modifier la base de données, de sorte qu'elle conserve la trace de telle ou telle information supplémentaire.

Si vous élaborez la base de données pour votre propre compte, vous vous heurterez probablement aux limitations de votre structure, car il est impossible de tout prévoir lors de la conception. Par exemple, vous allez peut-être recevoir des propositions depuis diverses contrées et donc devoir ajouter une colonne `PAYS`. Ou alors vous déciderez de conserver l'adresse de messagerie de vos contacts, et vous devrez insérer une colonne `COURRIEL`. Dans tous les cas, il vous faudra revenir sur la structure que vous avez créée. Dans cette section, je vais utiliser Access afin de modifier la table que je viens de créer. Notez que les autres outils RAD disposent de fonctionnalités semblables.

S'il devient nécessaire que vous mettiez à jour les tables de votre base de données, prenez un instant pour passer en revue tous les champs qu'elles

utilisent. Par exemple, vous pourriez en venir à ajouter un second champ ADRESSE pour les personnes dont l'adresse est complexe et un champ PAYS pour tenir compte des propositions émanant d'autres pays.

Quoiqu'il soit plutôt facile de mettre à jour les tables d'une base de données, vous devriez éviter de le faire autant que possible. Toute application qui dépend de la précédente structure de la base de données risque de ne plus pouvoir fonctionner et devra être modifiée. Si vous avez beaucoup d'applications de ce type, la tâche risque d'être très pénalisante. Essayez d'anticiper les évolutions nécessaires. Il vaut mieux passer un peu plus de temps sur la conception de la base de données que d'avoir à mettre à jour des applications écrites il y a des années. En effet, vous risquez d'avoir oublié comment elles fonctionnent, et de ne pas pouvoir les mettre à jour.

Pour insérer de nouvelles lignes et tenir compte d'évolutions, ouvrez la table et suivez ces étapes :

1. **Dans la fenêtre de création de la table, cliquez du bouton droit sur le petit carré coloré sur la gauche du champ VILLE pour sélectionner la ligne, et sélectionnez Insérer une ligne dans le menu qui apparaît.**

   Une ligne vierge apparaît au-dessus de l'emplacement du curseur, qui décale les lignes existantes, comme sur la Figure 4.8.

2. **Saisissez les champs que vous souhaitez ajouter à votre table.**

   J'ai ajouté le champ ADRESSE2 au-dessus du champ VILLE, et un champ PAYS au-dessus du champ TELEPHONE.

3. **Une fois que vous avez terminé vos modifications, enregistrez la table avant de la fermer.**

   Le résultat doit ressembler à la Figure 4.9.

Créer un index

Comme le nombre de propositions que vous recevez peut facilement se compter en centaines, vous aurez besoin d'un outil vous permettant d'isoler rapidement les enregistrements qui vous intéressent en fonction de différents critères. Supposons par exemple que vous souhaitiez regarder uniquement les propositions de toutes les personnes qui se prétendent votre frère. En partant de l'idée qu'aucun de vos frères n'a changé son nom pour des motifs professionnels ou futiles, il est facile de sérier les propositions correspondantes en vous basant sur la valeur du champ `Nom`, comme le montre la requête SQL ad hoc suivante :

```
SELECT * FROM POUVOIR
WHERE Nom = 'Marx' ;
```

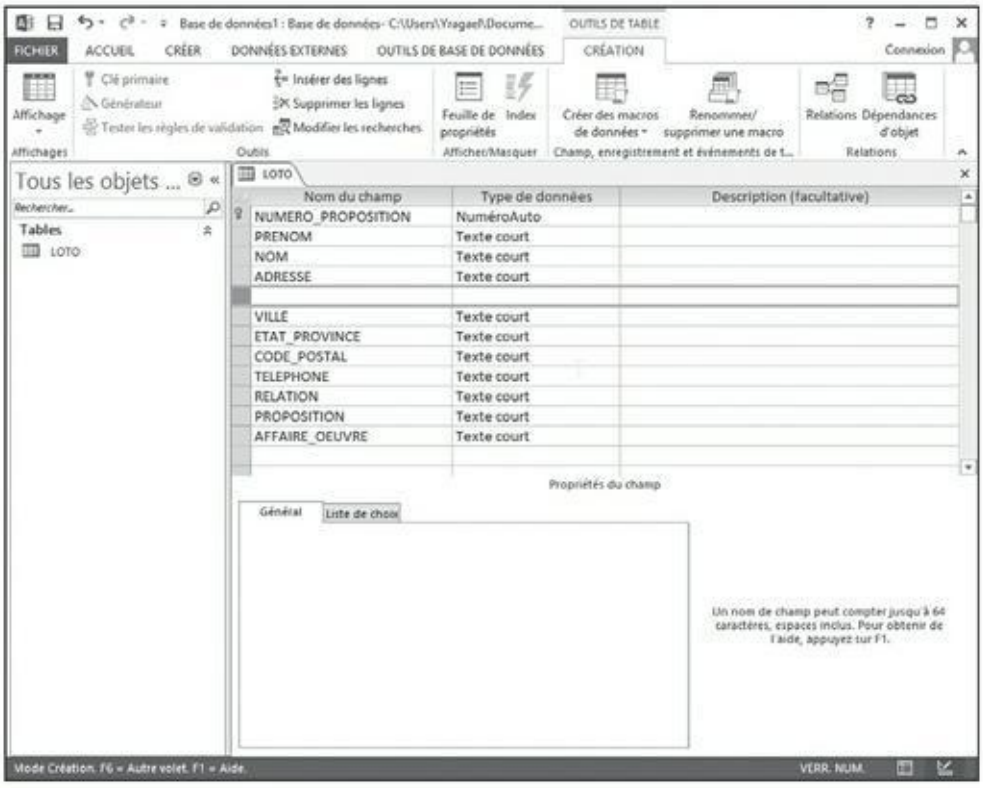


FIGURE 4.8 La fenêtre de création de la table après avoir rajouté un emplacement pour une seconde ligne d'adresse.

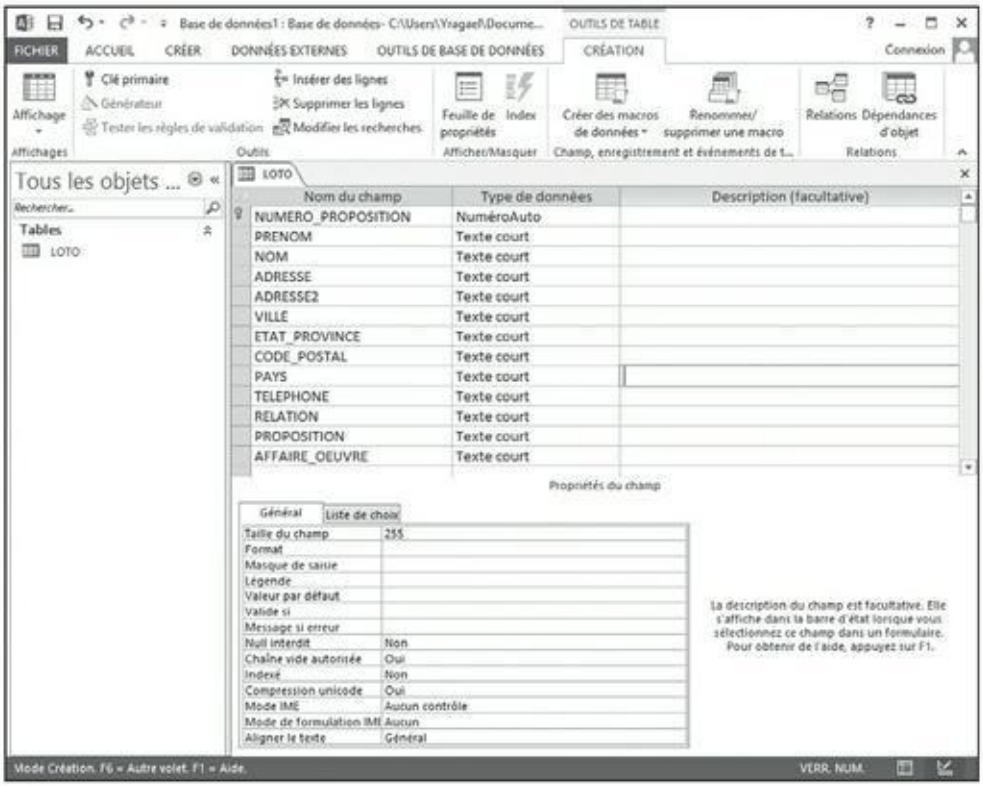


FIGURE 4.9 Votre table révisée devrait ressembler à ceci.

Cependant, cette stratégie ne peut pas s'appliquer aux propositions qui émanent de vos beaux-frères ou de vos belles-sœurs. Il est possible de se concentrer sur ces propositions en étudiant la valeur d'un autre champ :

```
SELECT * FROM POUVOIR
WHERE Relation = 'beau-frère'

        OR
        Relation = 'belle-soeur' ;
```

Toutes ces requêtes fonctionnent parfaitement, mais elles risquent de ne pas s'exécuter rapidement si POUVOIR contient beaucoup de données (des dizaines de milliers d'enregistrements). En effet, SQL passe la table en revue ligne par ligne, à la recherche des entrées qui pourraient satisfaire la clause WHERE. Vous pouvez considérablement accélérer les choses en associant un index à la table POUVOIR (un *index* est une table de pointeurs ; chaque ligne dans l'index pointe vers une ligne correspondante de la table).

Vous pouvez définir un index propre à chacune des manières par lesquelles vous comptez accéder à vos données. Si vous ajoutez, modifiez ou supprimez des lignes dans la table, vous n'aurez plus à la trier de nouveau : il vous suffira d'actualiser l'index. Et cette mise à jour s'effectue beaucoup plus rapidement que le tri d'une table. Une fois que vous disposerez d'un index correspondant au tri que vous souhaitez, vous pourrez l'utiliser pour accéder aux lignes de la table presque instantanément.

Comme `NUMERO_PROPOSITION` est unique et court, utiliser ce champ est le moyen le plus rapide pour accéder à un enregistrement donné. C'est donc un candidat idéal pour devenir clé primaire. Pour la même raison, *la clé primaire de chaque table et de toutes les tables devrait toujours être indexée*. Voilà qui tombe d'ailleurs bien, car Access indexe automatiquement les clés primaires ! Cependant, vous devez connaître la valeur `NUMERO_ PROPOSITION` de l'enregistrement que vous cherchez pour pouvoir accéder à ces données. C'est pourquoi vous pourriez créer des index basés sur d'autres champs tels que `NOM`, `CODE POSTAL` ou `RELATION`. Prenons un exemple concret. Dans le cas d'une table indexée sur le champ `Nom`, dès qu'une recherche trouve une ligne contenant `Marx`, elle a du même coup détecté tous les autres enregistrements ayant la même valeur dans la colonne Nom. Vous pouvez donc retrouver `Chico`, `Groucho`, `Harpo` et `Zeppo` pratiquement aussi vite que `Chico` seul.

La gestion d'un index provoque une charge de travail supplémentaire, et donc ralentit un peu votre système. Cependant, il permet d'accéder beaucoup plus rapidement aux données. Vous devez donc vous efforcer de trouver un bon compromis entre ces deux pôles contradictoires.

Voici quelques conseils pour vous aider à choisir les bons champs pour indexer votre table :

- Indexez les champs que vous utilisez le plus souvent. De cette manière, il sera possible d'accéder rapidement aux données sans que le surcroît de travail imposé au système ne pénalise inconsidérément vos recherches.
- Ne perdez pas votre temps à créer des index pour des champs qui ne vous servent jamais dans vos critères. Ce

serait un gaspillage inutile de temps et d'espace.

» Créer un index pour un champ dont la valeur ne permet pas d'identifier formellement une série cohérente d'enregistrements n'a généralement pas de sens. Par exemple, indexer le champ `AFFAIRE_OEUVRE` ne ferait que diviser votre table en deux catégories. Ce n'est donc pas un candidat intéressant.

L'efficacité d'un index varie d'une implémentation à une autre. Si vous faites migrer une base de données d'une plate-forme à une autre, il est possible que les index qui donnaient de bons résultats sur le premier système ne fonctionnent plus aussi bien sur le second. En fait, ils peuvent même pénaliser vos performances. Vous devez optimiser la gestion des index en fonction de chaque SGBD et de chaque configuration matérielle. Essayez différents schémas d'indexation pour juger de celui qui vous procure les meilleures performances d'ensemble. Optimisez vos index de manière à limiter au maximum leur impact négatif sur les recherches et la mise à jour des données.

Pour indexer la table POUVOIR, sélectionnez simplement Oui pour Indexé dans le panneau Propriétés du champ dans la fenêtre de création de la table.

Access crée automatiquement un index pour `CodePostal`, car ce champ est souvent utilisé pour récupérer des informations. Access indexe aussi automatiquement la clé primaire.

Vous pouvez voir que `CodePostal` n'est pas une clé primaire et n'est pas nécessairement unique. C'est exactement l'inverse pour `NUMERO_PROPOSITION`. Ajoutez des index supplémentaires pour `Nom` et Relation, car ces champs serviront probablement de critères lors de recherches.

Une fois que vous avez créé tous vos index, sauvegardez la nouvelle structure de la table avant de la fermer.

Si vous utilisez un autre RAD que Microsoft Access, la démarche décrite dans cette section ne s'applique pas à votre situation. Cependant, le processus général devrait être le même.

Supprimer une table

Au cours de la construction d'une table telle que POUVOIR, il se peut que vous passiez par quelques versions intermédiaires qui ne correspondent finalement pas à vos souhaits. La présence de ces tables inachevées peut induire vos futurs utilisateurs en erreur. Vous feriez donc mieux de les supprimer tant que vous savez à quoi elles correspondent. Pour ce faire, cliquez du bouton droit sur la table que vous souhaitez supprimer dans la liste Toutes les tables sur le côté gauche de la fenêtre. Un menu apparaît, et l'une des options qu'il propose est Supprimer. Lorsque vous cliquez sur Supprimer, comme le montre la Figure 4.10, la table est retirée de la base de données.

FIGURE 4.10 Sélectionnez Supprimer pour supprimer une table.

Si Access supprime une table, il supprimera aussi toutes celles qui en dépendent, ainsi que tous ses index.

Créer une table avec le DDL de SQL

Toutes les fonctions de définition de la base de données dont vous disposez via un outil RAD tel qu'Access peuvent aussi être effectuées en SQL. Plutôt que de cliquer dans des menus avec la souris, vous devrez saisir des commandes au clavier. Certaines personnes, qui préfèrent manipuler des objets visuels, pensent que les outils RAD sont plus faciles d'utilisation. D'autres, plus orientées vers la formulation de leur besoin par des séquences d'instructions logiques, trouvent que les instructions SQL sont plus simples.

Certains concepts se prêtent mieux à une représentation graphique, tandis que d'autres se gèrent plus directement sous SQL. Il est donc utile de bien maîtriser l'une et l'autre de ces méthodes.

Dans les sections suivantes, j'utilise SQL pour effectuer les opérations de création, modification et suppression de table que nous venons d'effectuer à l'aide du RAD.

Utiliser SQL avec Microsoft Access

Access est par essence un outil de développement rapide (RAD) qui ne nécessite pas de programmation. Il est évidemment possible d'écrire des instructions SQL sous Access, mais vous devrez pour cela passer par la porte de derrière… Voici comment ouvrir un éditeur (basique au demeurant) dans lequel vous saisirez votre code SQL :

1. **Ouvrez votre base de données et cliquez sur l'onglet Créer pour afficher le ruban en haut de l'écran.**

2. **Cliquez sur Création de requête dans la section Requêtes.**

   La boîte de dialogue Afficher la table apparaît.

3. **Sélectionnez la table LOTO. Cliquez sur le bouton Ajouter, puis sur Fermer pour fermer la boîte de dialogue.**

Vous devriez voir s'afficher ce qui est représenté sur la Figure 4.11.

Une image de la table LOTO et de ses attributs apparaît dans la partie supérieure de la zone de travail, et une grille Requête par exemple apparaît dessous. Access attend que vous saisissiez une requête en utilisant cette grille (vous *pourriez* le faire, c'est sûr, mais cela ne vous apprendrait pas à utiliser SQL dans l'environnement d'Access).

4. **Cliquez sur l'onglet Accueil puis sur l'icône Affichage dans l'angle gauche du ruban**

   Un menu apparaît, qui contient les différents modes auxquels vous pouvez accéder, comme sur la Figure 4.12.

   L'un de ces modes est la vue SQL.

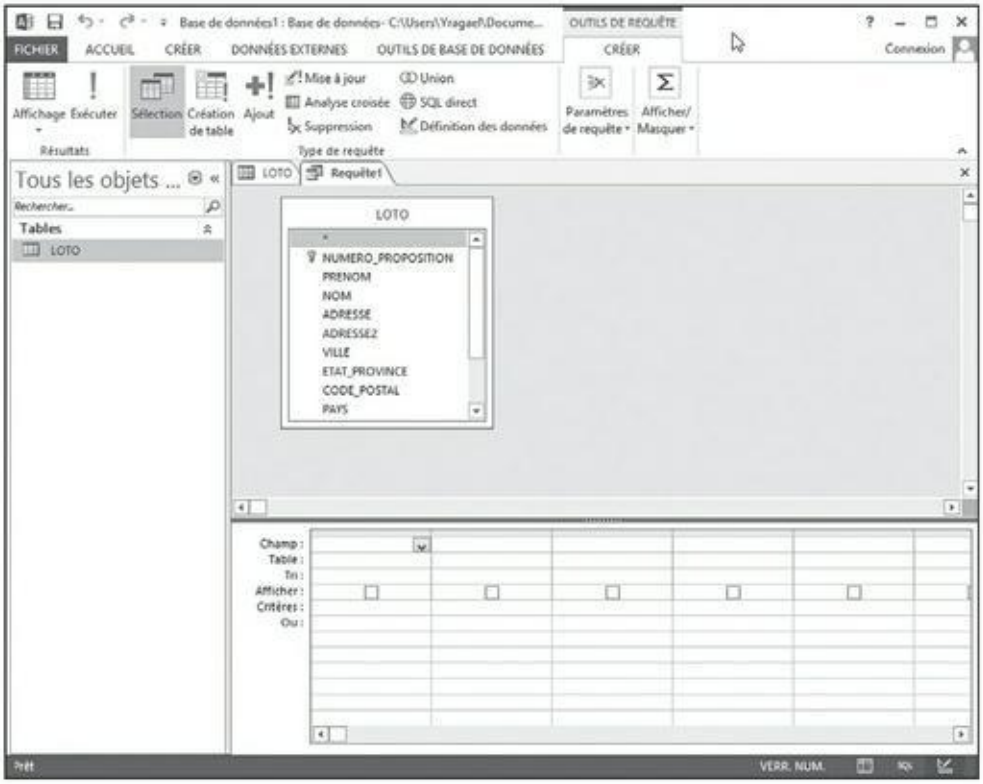


FIGURE 4.11 L'écran Requête une fois la table LOTO sélectionnée.

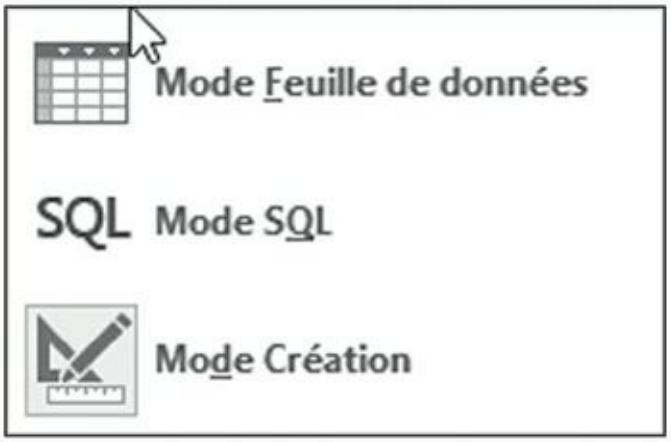


FIGURE 4.12 Les modes de la base de données disponibles en mode Requête.

5. **Cliquez sur Mode SQL pour afficher l'onglet de la requête en SQL.**

   Comme vous le voyez sur la Figure 4.13, l'onglet de la requête en SQL fait l'hypothèse (très rationnelle) que vous souhaitez récupérer des informations de la table LOTO, si bien qu'il a écrit la première partie de la requête pour vous. Il ne sait pas exactement ce que vous souhaitez récupérer, si bien qu'il n'affiche que ce dont il est certain.

   Voici ce qu'il a écrit pour vous :

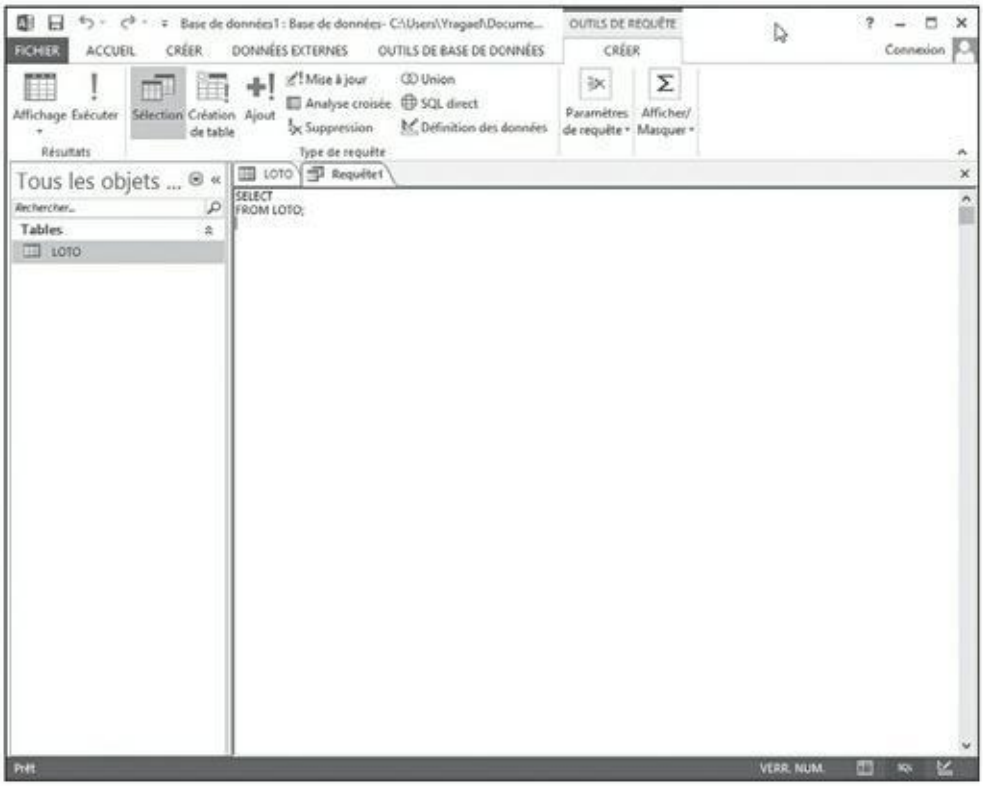


FIGURE 4.13 La vue SQL de l'onglet Objet.

```
SELECT
FROM POUVOIR ;
```

6. **Rajoutez un astérisque (*) à la fin de la première ligne et ajoutez une clause `WHERE` après la ligne `FROM`.**

   Si vous avez déjà saisi quelques données dans la table LOTO, vous pourriez les récupérer de cette manière :

```
SELECT *
FROM LOTO
       WHERE Prenom = 'Max' ;
```

   N'oubliez pas le point-virgule (`;`) qui termine la commande SQL. Vous devez le reporter de sa position après LOTO à la fin de la ligne suivante.

7. **Lorsque vous avez terminé, cliquez sur l'icône figurant une disquette.**

   Access vous demande de donner un nom à votre nouvelle requête.

8. **Attribuez un nom à votre instruction et cliquez OK.**

Votre requête est sauvegardée et pourra être exécutée à tout moment.

Créer une table

Pour créer une table de base de données en SQL (du moins avec un SGBD qui l'implante pleinement), il vous suffit de saisir les mêmes informations que celles que vous avez spécifiées en utilisant l'outil RAD. La différence est que l'outil RAD vous aide en affichant une boîte de dialogue du style Création de table et vous interdit d'entrer des noms de champs, des types et des longueurs qui ne sont pas valides.

SQL ne vous aide pas beaucoup. Vous devez savoir à l'avance exactement ce que vous faites, car il vous faudra saisir la totalité de l'instruction `CREATE TABLE` avant que SQL ne commence à la contrôler pour vérifier si elle contient ou non des erreurs. Jusque-là, vous êtes totalement seul.

L'instruction qui crée une table de suivi des propositions identique à celle que nous avons définie précédemment adopte la syntaxe suivante :

```
CREATE TABLE LOTO_SQL (
 NUMERO_PROPOSITION INTEGER PRIMARY KEY,
 PRENOM CHAR (15),
 NOM CHAR (20),
 ADRESSE CHAR (30),
 VILLE CHAR (25),
 ETAT_PROVINCE CHAR (2),
 CODE_POSTAL CHAR (10),
 PAYS CHAR (30),
 TELEPHONE CHAR (14),
 RELATION CHAR (30),
```

```
PROPOSITION CHAR (50),
AFFAIRE_OEUVRE CHAR (1) );
```

Comme vous pouvez le constater, l'information que contient l'instruction SQL est essentiellement la même que celle que vous avez saisie via l'outil RAD. Cependant, SQL présente l'avantage d'être un langage universel. La même syntaxe fonctionne sur tous les systèmes de gestion de bases de données.

Dans Access 2013, la création d'objets dans une base de données tels que les tables est un peu plus complexe. Vous ne pouvez pas simplement saisir la commande `CREATE` (comme vous venez de le voir) dans l'onglet de la requête en SQL. C'est parce que cet onglet ne sert que d'outil de requête ; vous devez procéder à quelques actions supplémentaires pour informer Access que vous souhaitez saisir une requête de définition de données plutôt qu'une requête normale qui ne fait que demander des informations figurant dans la base de données. Une autre complication : comme la création de table pourrait compromettre la sécurité de la base de données, elle est interdite par défaut. Vous devez indiquer à Access qu'il s'agit d'une base de données de confiance avant qu'il accepte la requête de définition de données.

1. **Cliquez sur l'onglet Créer dans le ruban pour afficher les icônes relatives à la création.**
2. **Cliquez sur le mode Création dans la section Requêtes.**

   Cela affiche la boîte de dialogue Afficher la table, qui contient pour l'heure plusieurs tables systèmes en plus de LOTO.
3. **Sélectionnez LOTO et cliquez sur le bouton Ajouter.**

   Comme vous l'avez vu dans l'exemple précédent, une image de la table LOTO et de ses attributs apparaît dans la moitié supérieure de l'écran.
4. **Cliquez sur le bouton Clone dans la boîte de dialogue Afficher la table.**

5. **Cliquez sur l'onglet Accueil, puis sur l'icône Affichage à gauche du ruban, et sélectionnez Mode SQL dans la liste déroulante qui apparaît.**

   Comme dans l'exemple précédent, Access vous a « aidé » en rajoutant `SELECT FROM LOTO` dans l'éditeur SQL. Cette fois, vous n'en voulez pas.

6. **Supprimez `SELECT FROM LOTO` et saisissez à la place la requête de définition de données présentée plus tôt, comme suit :**

```
CREATE TABLE LOTO_SQL (
       NUMERO_PROPOSITION INTEGER PRIMARY KEY,
       PRENOM CHAR (15),
       NOM CHAR (20),
       ADRESSE CHAR (30),
       VILLE CHAR (25),
       ETAT_PROVINCE CHAR (2),
       CODE_POSTAL CHAR (10),
       PAYS CHAR (30),
       TELEPHONE CHAR (14),
       RELATION CHAR (30),
       PROPOSITION CHAR (50),
       AFFAIRE_OEUVRE CHAR (1) );
```

   À cet instant, votre écran devrait ressembler à la Figure 4.14.

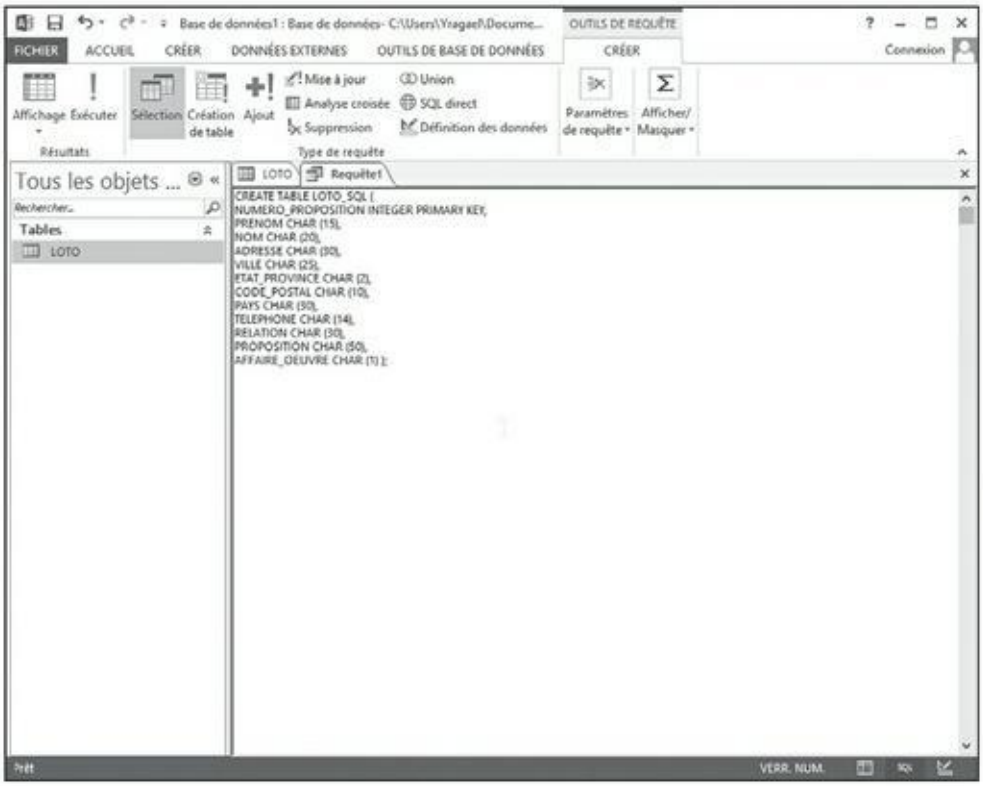


FIGURE 4.14 La requête de définition de données pour créer une table.

7. **Après avoir cliqué sur l'onglet Créer dans le ruban, cliquez sur le point d'exclamation de l'icône Exécuter.**

   Ce faisant, vous exécutez la requête, ce qui crée la table LOTO_SQL (comme sur la Figure 4.15).

   Vous devriez voir apparaître LOTO_SQL sous Tous les objets Access dans la colonne figurant sur le côté gauche de la page. Dans ce cas, vous avez de la chance. Ou alors, vous ne voyez pas la table dans la liste Tous les objets Access. Si c'est le cas, continuez de lire.

FIGURE 4.15 Création de la table LOTO_SQL.

Access 2013 fait de gros efforts pour vous protéger contre des hackers mal intentionnés et des utilisateurs imprudents. Comme exécuter une requête de définition de données peut constituer un danger pour la base de données, Access interdit par défaut l'exécution d'une telle requête. Si vous rencontrez cet obstacle, LOTO_SQL n'apparaîtra pas dans la colonne sur la gauche de la fenêtre car la requête n'aura pas été exécutée.

À la place, vous verrez apparaître ce message dans la barre de messages qui se trouve sous le ruban :

```
Alerte Sécurité: Du contenu dans la base de
données a
été désactivé.
```

Si vous voyez ce message, suivez ces étapes :

8. **Cliquez sur l'onglet Ficher, et dans le menu qui Figure sur le côté gauche de l'écran, sélectionnez Options.**

   La boîte de dialogue Options apparaît.

9. **Sélectionnez Centre de gestion de la confidentialité dans la base de données Options Access.**

10. **Cliquez sur le bouton Paramètres du centre de gestion de la confidentialité lorsqu'il apparaît.**

11. **Sélectionnez Barre de messages dans le menu sur la gauche, puis spécifiez Afficher la barre de messages en cliquant son bouton s'il n'est pas déjà sélectionné.**

12. **Cliquez pour revenir en arrière, là où vous avez tenté d'exécuter la requête de définition de données qui crée la table LOTO_SQL.**

13. **Exécutez la requête.**

Tous les efforts que vous consacrez à l'apprentissage de SQL seront payants à long terme, car SQL est parti pour durer. Le temps passé à devenir un expert d'un outil RAD est beaucoup moins rentable. Aussi sophistiqué que soit un outil RAD, vous pouvez être certain qu'il sera dépassé par une nouvelle technologie dans les cinq années à venir. Si vous pensez pouvoir amortir le coût de votre formation dans ce délai, parfait ! Formez-vous. Sinon, il serait sage de vous en tenir à SQL. Et la même remarque vaut pour l'ensemble de votre environnement de travail.

Créer un index

Les index sont très importants dans les bases de données. Ils servent de pointeurs dans des tables qui contiennent des données intéressantes. Si vous utilisez un index, vous pouvez accéder directement à un enregistrement particulier sans avoir à balayer séquentiellement la table, enregistrement

par enregistrement, à la recherche de celui que vous cherchez. Sans index, il vous faudrait peut-être attendre des années et non des secondes pour extraire une information d'une très grosse table. Des années, c'est peut-être beaucoup. Mais un temps suffisant pour que vous finissiez par vous lasser et abandonner la recherche, cela c'est certain.

D'une manière surprenante, le standard SQL ne propose aucune instruction pour créer un index. Les éditeurs de SGBD doivent donc écrire eux-mêmes cette fonctionnalité. Et comme les implémentations ne sont pas standardisées, elles diffèrent les unes des autres. La plupart des éditeurs proposent la création d'un index sous la forme d'une instruction `CREATE INDEX`.

Même si deux éditeurs utilisent les mêmes mots, cela ne signifie pas pour autant que ces deux commandes agissent de la même manière. Vous devrez probablement spécifier des clauses spécifiques à l'implémentation. Étudiez donc attentivement la documentation de votre SGBD pour élucider ce point.

Modifier la structure d'une table

Vous pouvez utiliser l'instruction `ALTER TABLE` pour modifier la structure d'une table qui existe déjà. Contrairement à l'outil RAD qui affiche la structure de cette table, SQL vous oblige à la connaître à l'avance pour la modifier. C'est évidemment moins commode. Par contre, l'utilisation de SQL est généralement la méthode la mieux adaptée si vous voulez inclure les instructions qui modifient la table dans un programme d'application.

Pour ajouter un second champ adresse à la table LOTO_SQL, utilisez l'instruction DDL suivante :

```
ALTER TABLE LOTO_SQL
ADD COLUMN ADDRESS_2 CHAR (30);
```

Vous n'avez pas besoin d'être un technogourou du SQL pour comprendre ce qu'elle veut dire. Cette instruction modifie une table appelée LOTO_SQL en y ajoutant une colonne. La colonne s'appelle ADRESSE_2, elle est de type `CHAR` (c'est donc une chaîne) et sa longueur est de 30 caractères. Cet exemple vous montre qu'il est très facile de modifier une table à l'aide de commandes DDL.

Le standard SQL fournit cette instruction pour ajouter une colonne. Il propose symétriquement une instruction vous permettant d'en supprimer une :

```
ALTER TABLE LOTO_SQL
DROP COLUMN ADDRESS_2;
```

Supprimer une table

Il est très facile de supprimer des tables dont vous n'avez plus besoin et qui encombrent votre disque dur :

```
DROP TABLE LOTO_SQL;
```

Comment cela pourrait-il être plus simple ? Si vous supprimez une table, vous détruisez aussi toutes ses données et ses métadonnées. Il n'en reste rien. Cela fonctionne bien, sauf si une autre table de la base de données fait référence à celle que vous essayez de supprimer. Cela s'appelle une *contrainte d'intégrité référentielle*. Dans ce cas, SQL vous produira un message plutôt que de supprimer la table.

Supprimer un index

Si vous supprimez une table en utilisant l'instruction DROP TABLE, vous supprimez aussi les index qui y sont associés. Cependant, il peut arriver que vous souhaitiez conserver une table mais supprimer un de ses index. Le standard SQL ne définit pas d'instruction du genre DROP INDEX. Par contre, vous la trouverez dans la plupart des implémentations. Cette commande se révèle fort utile quand votre système ralentit, que les utilisateurs commencent à réclamer de nouveaux ordinateurs plus puissants, et que vous découvrez que vos tables ne sont pas indexées de la meilleure manière.

De la portabilité

Toute implémentation de SQL que vous pourrez utiliser disposera très certainement d'extensions qui assurent des fonctionnalités qui ne sont pas

mentionnées dans le standard SQL. Certaines de ces fonctionnalités feront probablement leur apparition dans la prochaine version du standard. D'autres sont totalement spécifiques à certaines implémentations et le resteront vraisemblablement.

Ces extensions facilitent la création d'applications, si bien qu'il est tentant de les utiliser. Mais n'oubliez pas qu'y recourir présente aussi des inconvénients. Si vous voulez un jour migrer votre application vers une autre implémentation, vous devrez réécrire les sections de votre code dans lesquelles vous faites référence aux extensions que la nouvelle implémentation ne reconnaît pas. Pensez-y avant de faire appel aux extensions incluses dans votre SGBD actuel. Si elles vous font gagner du temps à court terme, elles peuvent aussi vous en faire perdre à plus long terme.

Plus vous en apprendrez sur les particularités de votre système, mieux vous serez à même de prendre les bonnes décisions. Par exemple, pour aboutir à la conclusion qu'il existe une autre solution à un problème x que de passer par des extensions de votre SGBD.

Chapitre 5

Créer une base de données relationnelle multitable

DANS CE CHAPITRE :

- **Décider de ce qu'il faut inclure dans une base de données.**
- **Déterminer les relations entre les données.**
- **Lier des tables à l'aide de clés.**
- **Préserver l'intégrité des données dès la conception.**
- **Normaliser une base de données.**

Dans ce chapitre, je vous présente un exemple de base de données qui comporte plusieurs tables. La première étape de la conception d'une telle base consiste à décider de ce qu'il faut y inclure et de ce qu'il faut en exclure. Les étapes suivantes visent à établir des relations entre les données et à configurer les tables en conséquence. Je vous explique aussi comment utiliser les clés, qui permettent d'accéder rapidement à des enregistrements spécifiques ainsi qu'à des index.

Une base ne doit pas simplement contenir vos données. Elle doit aussi les protéger contre toute forme de corruption. Dans la dernière partie de ce chapitre, je vous expliquerai comment préserver l'intégrité de vos données. La normalisation étant une des principales méthodes que vous pouvez utiliser à cette fin, je vous présenterai les diverses formes normales et les types de problèmes qu'elles peuvent permettre de résoudre.

Concevoir la base de données

Suivez les étapes suivantes pour concevoir la base (je reviendrai en détail sur chacune des étapes après les avoir toutes énoncées) :

1. **Identifiez les objets que vous souhaitez stocker dans votre base de données.**
2. **Déterminez parmi ces objets ceux qui devraient être des tables et ceux qui devraient être des colonnes dans ces tables.**
3. **Définissez les tables conformément à la manière dont vous avez besoin d'organiser les objets.**

   Vous pourriez éventuellement assigner à une ou plusieurs colonnes le rôle de clé. Les clés permettent de localiser rapidement dans une table la ligne qui vous intéresse.

Les sections suivantes reviennent dans le détail sur chacune de ces étapes.

Étape 1 : Identifier les objets

La première étape de la conception d'une base de données consiste à décider de ce qu'il est important d'intégrer au modèle. Traitez chaque aspect de la question sous forme d'un objet. Établissez une liste de tous les objets auxquels vous pouvez penser. N'essayez pas encore d'identifier les relations qui peuvent lier ces entités entre elles. Pour l'instant, essayez simplement de dresser une liste exhaustive.

Il est important de pouvoir bénéficier de l'aide de personnes qui connaissent bien le système que vous allez tenter de modéliser. Vous obtiendrez de précieuses informations en les incitant à parler entre elles et en les interrogeant lors d'une séance de brainstorming. En travaillant de manière collective, vous développerez probablement une collection d'objets plus complète et plus précise qu'en restant dans votre coin.

Une fois que vous disposez d'un ensemble d'objets qui vous semble raisonnablement complet, vous pouvez passer à l'étape suivante : décider comment ces objets seront reliés entre eux. Certains de ces objets sont des entités majeures, totalement indispensables pour fournir les résultats que

vous escomptez. D'autres sont subordonnés à ces entités principales. Enfin, certains objets n'ont peut-être finalement rien à faire dans votre modèle et devront être éliminés avant de venir « polluer » la base de données.

Étape 2 : Identifier les tables et les colonnes

Les entités majeures se transforment en tables. Chacune possède un ensemble d'attributs qu'il faut convertir en colonnes de la table. Par exemple, nombreuses sont les bases de données d'entreprise qui contiennent une table CLIENTS pour conserver la trace du nom et de l'adresse des clients (plus évidemment d'autres informations utiles). Chaque attribut d'un client, tel que son nom, sa rue, sa ville, son code postal et son adresse Internet, est transformé en une colonne de la table CLIENTS.

Il n'existe pas de règles générales vous aidant à identifier les objets qui devraient être des tables et à décider quels attributs ces tables doivent posséder. C'est à vous d'y réfléchir. Vous associerez un attribut à telle table pour une certaine raison, et tel autre attribut à telle autre table pour une raison différente. Basez votre décision sur ce que vous savez en respectant deux objectifs :

- Les informations que vous souhaitez pouvoir extraire de la base de données.
- Comment vous comptez utiliser les informations.

Lors de la conception de la structure des tables de votre base de données, il est extrêmement important de prendre en considération l'avis des futurs utilisateurs ainsi que celui des personnes qui prendront des décisions sur la base des informations qu'elle contiendra. Si la structure « raisonnable » à laquelle vous arrivez ne correspond pas à la manière dont les gens vont exploiter l'information, l'utilisation de votre système se révélera frustrante et pourra même déboucher sur des prises de décision erronées. De telles conséquences sont inacceptables !

Prenons un exemple pour illustrer le processus intellectuel qui débouche sur la création d'une base dotée de plusieurs tables. Supposons que vous venez de créer VetLab, un laboratoire de microbiologie qui teste des spécimens

que vous font parvenir des vétérinaires. Il est évident que vous allez devoir conserver la trace de nombreuses informations, et notamment :

- Les clients.
- Les tests que vous effectuez.
- Les employés.
- Les commandes.
- Les résultats.

Chacune de ces entités dispose d'attributs. Chaque client a un nom, une adresse et d'autres informations sont nécessaires pour gérer les contacts. Chaque test a un nom et un coût. Chaque employé a des coordonnées ainsi qu'un intitulé de poste et un salaire. Pour chaque commande, vous devez savoir qui l'a passée, quand elle a été passée et sur quel test elle portait. Pour chaque test, vous devez disposer d'un numéro d'ordre, savoir ce qu'il a donné, et si les résultats étaient préliminaires ou définitifs.

Étape 3 : Définir les tables

Vous devez maintenant définir une table par entité et une colonne par attribut. Le Tableau 5.1 présente les tables de VetLab.

TABLEAU 5.1 Les tables de VetLab.

Table	Colonnes
CLIENTS	Nom du client
	Adresse 1
	Adresse 2
	Ville
	Etat
	Code postal

	Téléphone
	Fax
	Contact
TESTS	Nom du test
	Prix standard
EMPLOYES	Nom de l’employé
	Adresse 1
	Adresse 2
	Ville
	Etat
	Code postal
	Téléphone personnel
	Code bureau
	Date d’embauche
	Classification
	Salarié horaire ou salarié ou commissionné
COMMANDES	Numéro de commande
	Nom du client
	Test
	Responsable commercial
	Date de commande
RESULTATS	Numéro

	Numéro de commande
	Résultat
	Date de production
	Préliminaire ou définitif

Vous pouvez créer ces tables à l'aide d'un outil RAD ou d'instructions DDL de SQL, de la manière suivante :

```
CREATE TABLSE CLIENTS (
NOM_CLIENT CHARACTER (30) NOT NULL,

ADRESSE_1 CHARACTER (30),
ADRESSE_2 CHARACTER (30),
VILLE CHARACTER (25),
ETAT CHARACTER (2),
CODE_POSTAL CHARACTER (10),
TELEPHONE CHARACTER (13),
FAX CHARACTER (13),
CONTACT CHARACTER (30) ) ;
CREATE TABLE TESTS (
NOM_TEST CHARACTER (30) NOT NULL,
PRIX_STANDARD CHARACTER (30) ) ;
CREATE TABLE EMPLOYES (
NOM_EMPLOYE CHARACTER (30) NOT NULL,
ADRESSE_1 CHARACTER (30),
ADRESSE_2 CHARACTER (30),
VILLE CHARACTER (25),
ETAT CHARACTER (2),
CODE_POSTAL CHARACTER (10),
TELEPHONE_PERSO CHARACTER (13),
CODE_BUREAU CHARACTER (4),
DATE_EMBAUCHE DATE,
```

```
CLASSIFICATION CHARACTER (10),

HEUR_SAL_COM CHARACTER (1) ) ;
CREATE TABLE COMMANDES (
NUMERO_COMMANDE INTEGER NOT NULL,
NOM_CLIENT CHARACTER (30),
TEST_COMMANDE CHARACTER (30),
COMMERCIAL CHARACTER (30),
DATE_COMMANDE DATE ) ;
CREATE TABLE RESULTATS (
NUMERO_RESULTAT INTEGER NOT NULL,
NUMERO_COMMANDE INTEGER,
RESULTAT CHARACTER(50),
DATE_PRODUCTION DATE,
PRELIMINAIRE_FINAL CHARACTER (1) ) ;
```

Ces tables sont reliées entre elles par les attributs (colonnes) qu'elles partagent, comme suit :

- La table CLIENTS est liée à la table COMMANDES par la colonne `NOM_CLIENT`.
- La table TESTS est liée à la table COMMANDES par la colonne `NOM_TEST(TEST_COMMANDE)`.
- La table EMPLOYES est liée à la table COMMANDES par la colonne `NOM_EMPLOYE(COMMERCIAL)`.
- La table RESULTATS est liée à la table COMMANDES par la colonne `NUMERO_COMMANDE`.

Pour qu'une table occupe vraiment un rôle dans une base de données relationnelle, il convient de la relier à une autre table par une colonne commune. La Figure 5.1 illustre les relations entre nos tables.

Les liens qui figurent sur la Figure 5.1 illustrent quatre relations un à plusieurs. Les losanges indiquent le lien de cardinalité entre chaque

extrémité d'une relation. Le chiffre 1 correspond au côté « un » des relations, et la lettre N au côté « plusieurs ».

- Un client peut passer plusieurs commandes, mais chaque commande est effectuée par un et un seul client.
- Chaque test peut figurer dans plusieurs commandes, mais chaque commande correspond à un et un seul test.
- Chaque commande est enregistrée par un et un seul employé (ou commercial), mais chaque commercial peut enregistrer plusieurs commandes (c'est préférable).

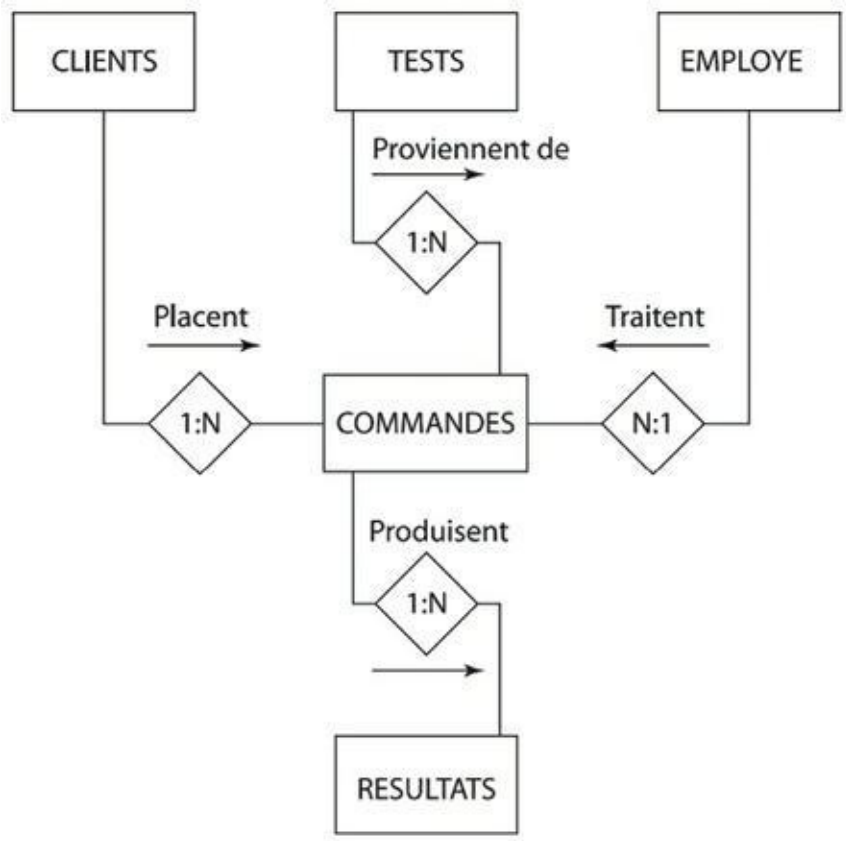


FIGURE 5.1 Les tables de la base de données de VetLab.

- Chaque commande peut déboucher sur plusieurs résultats préliminaires et sur un résultat final, mais chaque résultat n'est associé qu'à une et une seule commande.

Comme vous pouvez le constater sur la figure, l'attribut qui lie une table à une autre peut porter un nom différent dans ces tables. Cependant les deux attributs doivent avoir le même type de données.

Domaines, jeux de caractères, interclassements et translations

Bien que les tables soient les constituants principaux d'une base de données, celle-ci comporte d'autres éléments. Dans le Chapitre 1, j'ai défini le *domaine* d'une colonne d'une table comme étant l'ensemble des valeurs que peut prendre cette colonne. Une partie importante de la conception consiste à définir clairement des domaines de toutes les colonnes à l'aide de contraintes.

Les gens qui parlent anglais ne sont pas les seuls à utiliser des bases de données. Il est possible d'utiliser des langages dont le jeu de caractères est différent de celui de l'anglais. SQL vous permet de spécifier le jeu de caractères que vous souhaitez utiliser. En fait, vous pouvez même spécifier un jeu de caractères différent pour chaque colonne d'une table. SQL est l'un des très rares langages à offrir une telle souplesse.

Un *interclassement*, ou *séquence d'interclassement*, est un ensemble de règles qui spécifient comment les chaînes qui utilisent un certain jeu de caractères doivent être comparées entre elles. Chaque jeu de caractères est doté d'un interclassement par défaut. Dans le jeu de caractères ASCII, *A est avant B et B est avant C*. Une comparaison entre deux chaînes ASCII se basera donc sur le fait que *A est inférieur à B qui est inférieur à C*. SQL vous permet d'associer différents interclassements à un jeu de caractères. Là encore, il est l'un des rares langages capables de cela. C'est une raison supplémentaire pour aimer SQL.

Dans une base, vous encodez parfois des données à l'aide d'un jeu de caractères, mais il peut arriver que vous souhaitiez les traiter dans un autre jeu de caractères. Par exemple, vous disposez peut-être de données qui utilisent le jeu de caractères allemand, mais votre imprimante ne gère pas les caractères spécifiques à cette langue. Une traduction est une fonction de SQL qui vous permet de traduire des chaînes de caractères d'un jeu de caractères à un autre. Par exemple, la traduction pourrait convertir un caractère spécifique sous forme de deux lettres, comme le *ü* germanique en *ue* ASCII. Il est aussi possible de transformer des caractères minuscules en

majuscules. Vous pouvez même convertir un alphabet dans un autre, comme de l'arabe ou de l'hébreu en ASCII.

Améliorer les performances à l'aide des clés

Une bonne règle à appliquer dès la conception spécifie que chaque ligne d'une table doit pouvoir se distinguer de toutes les autres lignes de cette table, c'est-à-dire qu'elle doit être unique. Cette règle peut être transgressée aux tables que vous générez à l'occasion pour répondre occasionnellement à un besoin spécifique, comme par exemple produire des statistiques sur la base de critères précis. Mais elle doit s'appliquer à toutes les tables que vous comptez utiliser à des fins diverses.

Une *clé* est un attribut ou une combinaison d'attributs qui identifie de manière unique une ligne dans une table. Pour accéder à une certaine ligne, il faut que vous disposiez d'un procédé permettant de la distinguer des autres. Du fait qu'elles sont uniques, les clés constituent un excellent mécanisme d'accès.

Une clé ne doit jamais avoir une valeur nulle (c'est-à-dire vide ou non définie). En effet, si vous utilisiez des clés nulles, deux lignes qui contiendraient des valeurs nulles dans les champs de la clé ne pourraient plus se distinguer l'une de l'autre.

Dans mon exemple de laboratoire vétérinaire, vous pouvez assigner à certaines colonnes le rôle de clé. `NOM_CLIENT` ferait une bonne clé dans la table CLIENTS. Elle permettrait en effet de distinguer chaque client. NOM_TEST et `NOM_EMPLOYE` seraient de bonnes clés pour les tables TESTS et EMPLOYES. `NUMERO_ COMMANDE` et `NUMERO_RESULTAT` représenteraient de bonnes clés pour les tables COMMANDES et RESULTATS. Dans chaque cas, vérifiez que vous entrez une valeur unique pour chaque ligne.

Il existe deux sortes de clés : les clés primaires et les clés étrangères. Les clés dont je viens de parler sont des clés primaires. Ces clés garantissent l'unicité. Je vais parler des clés étrangères dans la section suivante.

Clés primaires

Une *clé primaire* est une colonne dont les valeurs identifient chaque ligne de la table d'une manière unique. Pour introduire cette notion dans la base de données VetLab, vous pouvez spécifier une clé primaire au moment où vous créez une table. Dans l'exemple suivant, une seule colonne suffit (en supposant que tous les clients de VetLab portent des noms différents) :

```
CREATE TABLE CLIENTS (
NOM_CLIENT CHARACTER (30) PRIMARY KEY,
ADRESSE_1 CHARACTER (30),
ADRESSE_2 CHARACTER (30),
VILLE CHARACTER (25),
ETAT CHARACTER (2),
CODE_POSTAL CHARACTER (10),
TELEPHONE CHARACTER (13),
FAX CHARACTER (13),
CONTACT CHARACTER (30)
)  ;
```

La contrainte `PRIMARY KEY` se substitue à la contrainte `NOT NULL` qui figurait dans la définition précédente de la table CLIENTS. Elle implique d'ailleurs la contrainte NOT NULL, car une clé primaire ne peut jamais prendre une valeur nulle.

Bien que la plupart des SGBD autorisent la création de tables sans clé primaire, toutes les tables d'une base devraient en posséder une. Pour appliquer ce concept, remplacez la contrainte `NOT NULL` dans toutes vos tables. Dans notre exemple, les tables TESTS, EMPLOYES, COMMANDES et RESULTATS vont recevoir une contrainte `PRIMARY KEY`, comme dans l'exemple suivant :

```
CREATE TABLE TESTS (
NOM_TEST CHARACTER (30) PRIMARY KEY,
PRIX_STANDARD CHARACTER (30) ) ;
```

Il arrive parfois qu'aucune colonne ne suffise à elle seule à garantir l'unicité. Vous devez alors utiliser une clé composite. Il s'agit d'une combinaison de colonnes qui, toutes ensemble, assurent cette unicité.

Supposez que certains des clients de VetLab soient des chaînes ayant des cabinets vétérinaires dans plusieurs villes. `NOM_CLIENT` n'est alors plus suffisant pour distinguer deux bureaux d'un même client. Vous pouvez alors régler le problème en définissant une clé composite de la manière suivante :

```
CREATE TABLE CLIENTS (
NOM_CLIENT CHARACTER (30) NOT NULL,
ADRESSE_1 CHARACTER (30),
ADRESSE_2 CHARACTER (30),
VILLE CHARACTER (25) NOT NULL,
ETAT CHARACTER (2),
CODE_POSTAL CHARACTER (10),
TELEPHONE CHARACTER (13),
FAX CHARACTER (13),
CONTACT CHARACTER (30),
CONSTRAINT BUREAU PRIMARY KEY
(NOM_CLIENT, VILLE)
)  ;
```

Clés étrangères

Une *clé étrangère* est une colonne, ou un groupe de colonnes, dans une table qui correspond à (ou référence) une clé primaire dans une autre table. Une clé étrangère n'a pas à être unique elle-même, mais elle doit identifier de manière unique la ou les colonnes qu'elle référence dans l'autre table.

Si la colonne `NOM_CLIENT` est une clé primaire dans la table CLIENTS, chaque ligne de cette table doit prendre une valeur unique dans la colonne `NOM_CLIENT`. `NOM_CLIENT` est une clé étrangère dans la table COMMANDES. Cette clé étrangère correspond à la clé primaire de la table CLIENTS, mais elle n'a pas à être unique dans la table COMMANDES. Et c'est tant mieux pour vous… Si chacun de vos clients vous passait une commande puis s'évanouissait dans la nature, votre laboratoire ne ferait pas long feu. Si tout va bien, de nombreuses lignes de la table COMMANDES vont correspondre à chaque ligne de la table CLIENTS, ce qui signifiera que presque tous vos clients sont fidèles à votre laboratoire.

La définition suivante de la table COMMANDES vous montre comment vous pouvez utiliser le concept de clé étrangère dans l'instruction CREATE :

```
CREATE TABLE COMMANDES (
NUMERO_COMMANDE INTEGER PRIMARY KEY,
NOM_CLIENT CHARACTER (30),

TEST_COMMANDE CHARACTER (30),
COMMERCIAL CHARACTER (30),
DATE_COMMANDE DATE,
CONSTRAINT NOM_CE FOREIGN KEY (NOM_CLIENT)
REFERENCES CLIENTS (NOM_CLIENT),
CONSTRAINT TEST_CE FOREIGN KEY (TEST_COMMANDE)
REFERENCES TESTS (NOM_TEST),
CONSTRAINT SALES_CE FOREIGN KEY (COMMERCIAL)
REFERENCES EMPLOYES (NOM_EMPLOYE)
)  ;
```

Dans cet exemple, les clés étrangères de la table COMMANDES lient celle-ci aux clés primaires des tables CLIENTS, TESTS et EMPLOYES.

Travailler avec des index

La spécification SQL ne mentionne pas les index, mais cette omission ne signifie pas pour autant que les index soient peu utilisés ni qu'ils soient en option dans un système de gestion de bases de données. Toutes les implémentations de SQL gèrent les index, mais d'une manière qui leur est propre. Dans le Chapitre 4, je vous ai montré comment vous pouviez créer un index en utilisant Microsoft Access, un outil de développement rapide d'applications (RAD). Vous devrez vous référer à la documentation de votre système de gestion de bases de données (SGBD) pour comprendre comment il implémente les index.

Qu'est-ce qu'un index ?

Les données que contient une table apparaissent généralement dans l'ordre dans lequel vous les avez saisies. Cependant, cet ordre ne correspond pas toujours à celui dans lequel vous comptez traiter les données. Par exemple, supposons que vous vouliez afficher votre table CLIENTS dans l'ordre de NOM_ CLIENT. L'ordinateur doit d'abord trier la table dans cet ordre. Plus la table est grande, plus ce tri prend de temps. Que se passera-t-il si votre table contient un million de lignes ? Il n'est pas rare que le cas se présente. Même les meilleurs algorithmes de tri devront effectuer vingt millions de comparaisons et des millions d'interversions pour trier la table. Même sur un ordinateur très rapide, vous allez trouver le temps bien long.

Les index vous permettent de gagner du temps. Un index est une table subsidiaire qui est liée à une table de données. Pour chaque ligne de cette dernière, il existe une ligne correspondante dans la table d'index. Cependant, l'ordre de ces lignes est différent dans l'index.

Le Tableau 5.2 vous montre un exemple de table de données.

TABLEAU 5.2 Table CLIENTS.

NOM_CLIENT	ADRESSE_1	ADRESSE_2	VILLE	ETAT
Clinique Butternut Animal	5, Butternut Lane		Hudson	NH
Amber Veterinary, Inc	470 Kolvir Circle		Amber	MI
Vets R Us	2300 Geoffrey Road	Suite 230	Anaheim	CA
Docteur Doggie	32 Terry Terrace		Nutley	NJ
Centre équestre	Département vétérinaire	7890 Paddock Parkway	Gallup	NM
Institut océanographique	1002 Marine Drive		Key West	FL

J.C. Campbell, Vétérinaire	2500 Maon Street		Los Angeles	CA
Ferme à vers de Wenger	15 Bait Boulevard		Sedons	AZ

Les lignes ne sont pas triées par ordre alphabétique selon NOM_CLIENT. En fait, elles ne sont pas triées du tout. Elles sont simplement dans l'ordre dans lequel quelqu'un les a saisies.

Un index pour la table CLIENTS peut ressembler au Tableau 5.3.

TABLEAU 5.3 L'index par nom de client pour la table CLIENTS.

NOM_CLIENT	Pointeur vers la table de données
Amber Veterinary, Inc	2
Clinique Butternut Animal	1
Docteur Doggie	4
Institut océanographique	6
J.C. Campbell, Vétérinaire	7
Centre équestre	5
Vets R Us	3
Ferme à vers de Wenger	8

L'index contient le champ sur lequel est basé l'index (ici `NOM_ CLIENT`) et un pointeur vers la table des données. Ce pointeur donne le numéro de la ligne correspondante dans la table de données.

Pourquoi utiliser un index ?

Si je veux traiter une table dans l'ordre `NOM_CLIENT`, et que je dispose d'un index arrangé selon cet ordre, je peux effectuer toutes mes opérations presque aussi vite que si les données avaient été saisies en classant alphabétiquement les noms des clients. Il est possible de parcourir l'index séquentiellement, et donc de récupérer ainsi successivement chaque pointeur vers la donnée qui lui correspond dans la table.

Si vous utilisez un index, le temps pris pour traiter la table sera proportionnel à N, où N est le nombre d'enregistrements dans la table. Sans index, ce temps sera proportionnel à $N \lg N$, où N est le logarithme base 2 de N. La différence n'est pas significative si la table est petite, mais elle le devient dès lors que sa taille est volumineuse. Certaines opérations ne peuvent être envisagées sans indexer les tables.

Supposons par exemple que vous disposiez d'une table qui contient 1 000 000 d'enregistrements (N = 1 000 000) et que traiter chaque enregistrement prenne une milliseconde (un millième de seconde). Si vous utilisez un index, le traitement de la table entière demandera 1 000 secondes, soit à peu près 17 minutes. Sans index, ce traitement prendra 1 000 000 x 20 millisecondes, soit 20 000 secondes, c'est-à-dire à peu près 5 heures et demie. Je pense que vous conviendrez du fait que la différence est substantielle.

Tenir un index à jour

Une fois que vous avez créé un index, vous devez le tenir à jour. Fort heureusement, votre SGBD assurera cette maintenance chaque fois que vous modifierez la table de données associée. Ce travail prend un peu de temps, mais cela en vaut la peine. Lorsque vous aurez créé un index et que votre SGBD le maintiendra à jour, cet index sera disponible à tout instant pour accélérer le traitement de données.

Le meilleur moment pour définir un index est celui où vous créez la table à laquelle il est associé. De cette manière, vous n'aurez pas à souffrir de la longue attente inhérente à la création d'un index quand une table est déjà remplie de données. Essayez toujours de prévoir à l'avance vos besoins en matière d'accès aux données. Définissez alors un index *pour chaque possibilité*.

Certains SGBD vous permettent de désactiver la maintenance des index. Il peut être utile de le faire dans le contexte d'applications temps réel où cette

mise à jour consomme trop de temps pour le peu de disponibilités dont vous disposez (« Désolé, chef, il faut arrêter le compte à rebours de la navette parce que les index ne sont pas à jour ! »). Il arrive parfois que vous puissiez programmer la mise à jour des index à des heures où la base est peu sollicitée.

Ne tombez pas dans le piège d'ajouter un index pour optimiser des recherches auxquelles vous allez rarement procéder. Gérer un index prend du temps, car c'est une opération supplémentaire que l'ordinateur doit accomplir chaque fois qu'il modifie le champ indexé, qu'il ajoute ou qu'il supprime une ligne. Ne créez des index que pour des tables de taille importante, et pour optimiser des recherches auxquelles vous comptez procéder souvent. Sinon, les performances de votre système s'en trouveront dégradées.

Il peut arriver que vous ayez à réaliser un rapport annuel ou trimestriel dont la production prendrait trop de temps si elle ne s'appuyait pas sur un index (qui ne vous sert d'ailleurs qu'à cette occasion). Créez cet index juste auparavant, produisez le rapport, puis supprimez l'index de sorte qu'il ne perturbe pas le travail du SGBD jusqu'à la prochaine demande de rapport.

Maintenir l'intégrité

Une base de données n'est utile que si vous êtes certain que les données qu'elle contient sont correctes. Par exemple, la présence de données erronées dans une base de données médicale, aéronautique ou militaire peut déboucher sur la mort d'autrui. Le concepteur d'une base de données devrait être en permanence certain qu'il est impossible à des données invalides de pénétrer la base.

Certes, cet objectif n'est pas toujours réalisable en totalité. Par contre, il est toujours possible de s'assurer (au moins) que les valeurs saisies sont valides. Maintenir *l'intégrité des données* signifie s'entourer de toutes les garanties pour que tout ce qui est entré dans la base satisfasse aux contraintes qui ont été établies auparavant. Si, par exemple, un certain champ est du type `Date`, le SGBD devrait y rejeter toute tentative de saisie d'une valeur autre qu'une date valide.

Certains problèmes ne peuvent pas être stoppés au niveau de la base de données. Le programmeur de l'application doit les intercepter avant qu'ils n'endommagent la base de données. Toute personne chargée de manipuler la

base de données doit bien être consciente des dangers qui menacent l'intégrité de cette dernière et prendre les mesures qui s'imposent pour désactiver ces failles.

L'intégrité d'une base de données peut relever de plusieurs types. Et de nombreux problèmes sont susceptibles d'affecter cette intégrité. Dans les sections suivantes, je traite de trois types d'intégrité : *l'intégrité de l'entité*, *l'intégrité du domaine* et *l'intégrité référentielle*. Nous verrons également quelques-unes des menaces qui planent sur cette intégrité.

L'intégrité de l'entité

Chaque table d'une base de données correspond à une entité du monde réel. Cette entité peut être physique ou conceptuelle, mais, d'une certaine manière, son existence est indépendante de celle de la base. L'intégrité de l'entité d'une table est assurée quand la table est totalement cohérente avec l'entité qu'elle représente. Pour cela, la table doit disposer d'une clé primaire. Celle-ci identifie chaque ligne de la table. Sans clé primaire, vous ne pouvez pas être certain qu'une ligne récupérée soit la bonne.

Pour maintenir cette intégrité, vous devez spécifier que la colonne ou le groupe de colonnes qui constitue la clé primaire est `NOT NULL`. De plus, vous devez contraindre la clé primaire pour qu'elle soit `UNIQUE`. Certaines implémentations de SQL vous permettent de mentionner cette contrainte lors de la déclaration de la table. D'autres ne le permettent pas, si bien que vous devez la spécifier après avoir spécifié comment ajouter, modifier et supprimer des données d'une table. La meilleure solution pour vous assurer que votre clé primaire est à la fois `NOT NULL` et `UNIQUE` est de spécifier la contrainte `PRIMARY KEY` lorsque vous créez la table, comme dans l'exemple suivant :

```
CREATE TABLE CLIENTS(
NOM_CLIENT CHARACTER (30) PRIMARY KEY,
ADRESSE_1 CHARACTER (30),
ADRESSE_2 CHARACTER (30),
VILLE CHARACTER (25),
ETAT CHARACTER (2),
CODE_POSTAL CHARACTER (10),
```

```
TELEPHONE CHARACTER (13),
FAX CHARACTER (13),
CONTACT CHARACTER (30)
)  ;
```

Une autre solution consiste à utiliser `NOT NULL` avec `UNIQUE`, comme dans l'exemple suivant :

```
CREATE TABLE CLIENTS (
NOM_CLIENT CHARACTER (30) NOT NULL,
ADRESSE_1 CHARACTER (30),
ADRESSE_2 CHARACTER (30),
VILLE CHARACTER (25),
ETAT CHARACTER (2),

CODE_POSTAL CHARACTER (10),
TELEPHONE CHARACTER (13),
FAX CHARACTER (13),
CONTACT CHARACTER (30)
)  ;
```

L'intégrité du domaine

En général, vous ne pouvez pas garantir qu'une donnée particulière de la base de données sera correcte, mais vous pouvez du moins vous assurer de sa validité. De nombreuses données ne peuvent prendre qu'un certain nombre de valeurs. Si vous remarquez une entrée qui ne prend pas une de ces valeurs possibles, c'est qu'elle est erronée. Par exemple, les États-Unis possèdent 50 États plus le district de Columbia, Puerto Rico et quelques autres possessions. Chacune de ces régions est désignée par un code à deux lettres reconnu par les services postaux. Si votre base de données contient une colonne `ETAT`, vous pouvez assurer l'intégrité du domaine en spécifiant que toute entrée dans la colonne `ETAT` doit être l'un de ces codes à deux lettres. Si quelqu'un saisit une valeur qui ne fait pas partie de cette liste, il enfreint l'intégrité des données. Si vous testez l'intégrité de votre domaine,

vous pouvez refuser d'accepter toute opération qui provoque ce genre de brèche.

L'intégrité du domaine peut être menacée si vous ajoutez une nouvelle donnée à une table en utilisant l'une des instructions `INSERT` ou `UPDATE`. Vous pouvez spécifier le domaine d'une colonne en utilisant l'instruction `CREATE DOMAIN` avant d'utiliser cette colonne dans une instruction `CREATE TABLE`, comme dans l'exemple suivant :

```
CREATE DOMAIN LEAGUE_DOM CHAR (8)
CHECK (LEAGUE IN ('Américaine', 'Nationale'));
CREATE TABLE TEAM (
TEAM_NAME CHARACTER (20) NOT NULL,
LEAGUE CHARACTER (8) NOT NULL
)  ;
```

Le domaine de la colonne `LIGUE` contient seulement deux valeurs valides : `Américaine` et `Nationale`. Votre SGBD refusera de valider la saisie ou de modifier une ligne de la table `EQUIPES` si la colonne `LIGUE` ne prend pas l'une de ces deux valeurs.

L'intégrité référentielle

Même si l'intégrité de l'entité et du domaine de chaque table de votre système est assurée, vous pouvez toujours être confronté à des problèmes parce que les relations qui relient vos tables sont incohérentes. Dans la plupart des bases de données bien conçues, chaque table contient au moins une colonne qui se réfère à une colonne dans une autre table. Ces références sont importantes pour maintenir l'intégrité de la base de données. Pour autant, elles favorisent l'apparition d'anomalies lors des modifications.

Les *anomalies de modification* sont des problèmes qui se produisent quand vous modifiez une donnée dans une ligne d'une table.

Les relations entre les tables ne sont généralement pas bidirectionnelles. Une table dépend normalement de l'autre. Supposons par exemple que vous disposiez d'une base de données qui contient des tables CLIENTS et

COMMANDES. Vous pourriez saisir un client dans la table CLIENTS avant qu'il ait passé une commande. Cependant, vous ne pouvez pas saisir une commande dans COMMANDES tant que le client n'a pas été créé dans la table CLIENTS. Ce type de relation est nommé relation parent-enfant, la table CLIENTS étant la table parent et la table COMMANDES la table enfant. L'enfant dépend du parent.

Généralement, la clé primaire de la table parent est une colonne (ou un groupe de colonnes) qui apparaît dans la table enfant. À l'intérieur de cette table enfant, elle (la colonne) ou il (le groupe de colonnes) est une clé étrangère. Une clé étrangère peut prendre une valeur nulle et n'a pas à être unique.

Les anomalies dans les modifications surviennent de plusieurs manières. Par exemple, un client déménage et vous voulez le retirer de votre base de données. S'il a déjà passé des commandes (enregistrées) dans la table COMMANDES, cette suppression va engendrer des problèmes. En effet, il existera alors des enregistrements dans la table COMMANDES (enfant) pour lesquels il n'existe pas d'enregistrement dans la table CLIENTS (parent). Des problèmes de cette nature peuvent aussi survenir si vous ajoutez un enregistrement à une table enfant sans effectuer les ajouts correspondants à la table parent.

Les clés étrangères de toute table enfant doivent refléter les modifications apportées à la clé primaire correspondante de la table parent, sans quoi il y aura anomalie de modification.

Les suppressions en cascade – à utiliser avec soin

Vous pouvez vous prémunir contre presque tous les problèmes d'intégrité référentielle en contrôlant avec soin le processus de mise à jour. Dans certains cas, vous devez répercuter en cascade les suppressions de la table parent dans ses tables enfants. Si vous supprimez une ligne de la table parent, vous devez alors supprimer toutes les lignes de ses tables enfants qui contiennent une clé étrangère correspondant à la clé primaire que vous venez de supprimer dans la table parent. Prenons un exemple pour mieux comprendre ce processus :

```
CREATE TABLE CLIENTS(
NOM_CLIENT CHARACTER (30)   PRIMARY KEY,
```

```
ADRESSE_1 CHARACTER (30),
ADRESSE_2 CHARACTER (30),
VILLE CHARACTER (25)        NOT NULL,
ETAT CHARACTER (2),
CODE_POSTAL CHARACTER (10),
TELEPHONE CHARACTER (13),
FAX CHARACTER (13),
CONTACT CHARACTER (30)
)  ;

CREATE TABLE TESTS (
NOM_TEST CHARACTER (30)     PRIMARY KEY,
PRIX_STANDARD CHARACTER (30)
)  ;

CREATE TABLE EMPLOYES (
NOM_EMPLOYE CHARACTER (30)  PRIMARY KEY,
ADRESSE_1 CHARACTER (30),
ADRESSE_2 CHARACTER (30),
VILLE CHARACTER (25),
ETAT CHARACTER (2),
CODE_POSTAL CHARACTER (10),
TELEPHONE_PERSO CHARACTER (13),
CODE_BUREAU CHARACTER (4),
DATE_EMBAUCHE DATE,
CLASSIFICATION CHARACTER (10),
HEUR_SAL_COM CHARACTER (1)
)  ;

CREATE TABLE COMMANDES (
NUMERO_COMMANDE INTEGER     PRIMARY KEY,
NOM_CLIENT CHARACTER (30),
TEST_COMMANDE CHARACTER (30),
```

```
COMMERCIAL CHARACTER (30),
DATE_COMMANDE DATE
CONSTRAINT NOM_CE FOREIGN KEY (NOM_CLIENT)
REFERENCES CLIENTS (NOM_CLIENT)
ON DELETE CASCADE,
CONSTRAINT TEST_CE FOREIGN KEY (TEST_COMMANDE)

          REFERENCES TESTS (NOM_TEST)
                    ON DELETE CASCADE,
CONSTRAINT VENTES_CE FOREIGN KEY (COMMERCIAL)
          REFERENCES EMPLOYES (NOM_EMPLOYE)
                    ON DELETE CASCADE
)  ;
```

La contrainte `NOM_CE` désigne `NOM_CLIENT` comme clé étrangère qui référence la colonne `NOM_CLIENT` dans la table CLIENTS. Si vous supprimez une ligne dans la table CLIENTS, vous détruisez alors automatiquement toutes les lignes de la table COMMANDES dont la colonne `NOM_CLIENT` contient la même valeur que la colonne équivalente de la table CLIENTS. La suppression se déroule en cascade depuis la table CLIENTS jusqu'à la table COMMANDES. Il en va de même pour les clés étrangères dans la table COMMANDES qui se réfèrent à des clés primaires dans les tables TESTS et EMPLOYES.

D'autres manière de contrôler les anomalies de mise à jour

Vous pourriez préférer à cette suppression en cascade une autre solution, qui consiste à passer la valeur `NULL` dans les clés étrangères d'une table enfant. Considérez la variante suivante de l'exemple précédent :

```
CREATE TABLE COMMANDES (
NUMERO_COMMANDE INTEGER     PRIMARY KEY,
NOM_CLIENT CHARACTER (30),
TEST_COMMANDE CHARACTER (30),
```

```
COMMERCIAL CHARACTER (30),
DATE_COMMANDE DATE
CONSTRAINT NOM_CE FOREIGN KEY (NOM_CLIENT)
REFERENCES CLIENTS (NOM_CLIENT)
CONSTRAINT TEST_CE FOREIGN KEY (TEST_COMMANDE)
REFERENCES TESTS (NOM_TEST)
CONSTRAINT VENTES_CE FOREIGN KEY (COMMERCIAL)
REFERENCES EMPLOYES (NOM_EMPLOYE)
 )  ;
```

La contrainte `VENTES_CE` désigne la colonne COMMERCIAL comme clé étrangère référençant la colonne `NOM_EMPLOYE` de la table EMPLOYES. Si un commercial quitte la société, vous supprimez sa ligne dans la table EMPLOYES. Un nouvel employé prendra probablement la place de celui qui vient de partir, mais pour l'instant la suppression du nom du commercial de la table EMPLOYES entraîne le passage à `NULL` de la colonne COMMERCIAL des commandes qu'il traitait.

Il est aussi possible d'empêcher l'intrusion de données invalides ou inconsistantes dans une base en utilisant l'une de ces méthodes :

- **Refusez d'autoriser tout ajout dans une table enfant tant qu'il n'existe pas de ligne correspondante dans la table parent**. Vous éviterez ainsi l'ajout de lignes « orphelines » dans la table enfant. De cette manière, vous aiderez à conserver des tables cohérentes.
- **Refusez que la clé primaire d'une table puisse être modifiée**. De cette manière, vous n'aurez plus à mettre à jour des clés étrangères dans d'autres tables qui dépendent de cette clé primaire.

Le jour où vous avez pensé que

tout était fini…

La seule chose dont on peut être sûr dans la vie réelle, c'est du changement. Dans le monde des bases de données, c'est pareil. Vous avez créé une base, vous avez défini vos tables, vos contraintes, vous avez rempli des lignes et des lignes de données… Et c'est à cet instant que l'ordre tombe d'en haut : la structure doit être changée. Comment ajouter une nouvelle colonne à une table existante ? Comment en supprimer une qui est devenue inutile ? Ne paniquez pas : SQL est là !

Ajouter une colonne à une table existante

Supposons que votre société décide que chaque anniversaire d'un membre du personnel devra être fêté. Un coordinateur est nommé à cet effet. Pour lui permettre de planifier ces petites sauteries, vous devez ajouter une colonne `DATE_NAISSANCE` à la table EMPLOYES. Pas de problème ! Il vous suffit de faire appel à l'instruction `ALTER TABLE`. Voici comment :

```
ALTER TABLE EMPLOYES
ADD COLUMN Date_Anniversaire AS DATE ;
```

Il n'y a plus qu'à saisir la bonne date de naissance dans chaque ligne de la table, et c'est parti pour les agapes…

Supprimer une colonne d'une table existante

Les affaires vont mal, et votre entreprise n'a plus les moyens d'offrir une petite fête à ses personnels le jour de leur anniversaire. Plus de champagne, plus de dates de naissance… Là encore, l'instruction ALTER TABLE vous permet de faire face à cette douloureuse situation :

```
ALTER TABLE EMPLOYES
DROP COLUMN Date_Anniversaire ;
```

Au moins, vous en avez profité tant que les affaires marchaient bien !

Sources potentielles de problèmes

L'intégrité de votre base de données est menacée par toutes sortes de problèmes. Certains d'entre eux ne se posent qu'avec des bases de données qui contiennent plusieurs tables, tandis que d'autres peuvent surgir même avec des bases de données ne comprenant qu'une seule table.

Mauvaise saisie

Les documents ou les fichiers que vous utilisez pour alimenter votre base de données peuvent contenir des données erronées (qu'il s'agisse d'informations valides mais qui ont été corrompues ou de valeurs parfaitement indésirables). Des tests aux limites vous permettront de vérifier que la donnée ne menace pas l'intégrité du domaine. Cependant, ce type de test ne prévient pas tous les problèmes. Les valeurs qui se trouvent dans les limites autorisées mais qui sont néanmoins erronées ne seront pas détectées.

Erreur de l'opérateur

La donnée à saisir peut être correcte, mais l'opérateur la transcrit mal. Ce type d'erreur peut conduire aux mêmes problèmes que ceux posés par l'utilisation de données erronées. Là encore, un test aux limites peut prévenir en partie cette erreur. Une meilleure solution serait qu'un second opérateur valide les saisies du premier. Cependant, c'est une solution coûteuse, car vous devrez doubler à la fois vos effectifs et le temps de saisie. Mais dans certaines circonstances ce sacrifice peut parfois vous éviter de perdre plus qu'il ne vous coûte.

Panne mécanique

Si une panne mécanique survient, telle que le crash d'un disque quand une table est ouverte, les données qui se trouvent dans la table peuvent être détruites. Votre seule solution est de réaliser des sauvegardes fréquentes.

Malveillance

Quelqu'un pourrait vouloir volontairement corrompre vos données. Votre première ligne de défense sera de refuser l'accès à votre base à quiconque

pourrait faire preuve de malveillance et de limiter les droits des autres personnes au strict nécessaire. Votre seconde ligne de défense sera de stocker des sauvegardes en lieu sûr. Testez régulièrement la sécurité de votre installation. Cela ne fait pas de mal d'être un peu paranoïaque.

Redondance de données

La *redondance de données* est un problème connu du modèle de base de données hiérarchique, mais il peut aussi affecter des bases de données relationnelles. Cette redondance gaspille de l'espace de stockage, ralentit les traitements et peut corrompre sérieusement les données. Si vous stockez la même donnée dans deux tables différentes, une modification de la copie de cette donnée dans une table peut ne pas être répercutée dans l'autre. Il y a alors incohérence puisqu'il n'est pas toujours possible de dire quelle est la bonne information. Il faut donc limiter autant que faire se peut la redondance des données.

Il faut certes un peu de redondance, ne serait-ce que pour que la clé primaire d'une table serve de clé étrangère dans une autre table, mais pas trop. L'une des solutions les plus efficaces pour limiter la redondance est la normalisation, qui consiste à découper une table en plusieurs tables plus simples.

Une fois que vous aurez éliminé la redondance de votre modèle de données, vous réaliserez peut-être que les performances sont insuffisantes. Les opérateurs ont souvent recours à la redondance pour accélérer les traitements. Dans l'exemple précédent de la base de VetLab, la table COMMANDES contient seulement le nom du client pour identifier la source de chaque commande. Si vous préparez une commande, vous devez effectuer la jointure des tables CLIENTS et COMMANDES afin d'obtenir l'adresse du client. Si cette jointure vous semble prendre trop de temps, vous pourriez stocker l'adresse du client dans la table COMMANDES. La redondance ainsi créée permet de ne plus recourir à la jointure et accélère donc la préparation d'une commande. En contrepartie, l'efficacité globale du système sera légèrement dégradée, et la mise à jour des adresses des clients sera plus délicate.

Les utilisateurs conçoivent souvent des bases de données avec un minimum de redondance et un haut degré de normalisation. Lorsqu'ils constatent que les performances d'applications importantes sont mauvaises, ils décident de dénormaliser leurs bases et d'y introduire sélectivement une certaine redondance. Le mot important est ici *sélectivement*. Vous devez prendre des

mesures appropriées pour que la redondance ne provoque pas plus de problèmes qu'elle n'en résout. Pour plus d'informations, voyez plus loin la section consacrée à la normalisation.

Dépassement de la capacité du SGBD

Un système de bases de données peut fonctionner parfaitement pendant des années. Puis des erreurs commencent à se produire de temps à autre. Et ces problèmes deviennent de plus en plus sérieux. C'est peut-être le signe que vous approchez des limites de votre système. Le nombre des lignes contenues dans une table n'est en aucun cas infini. De même, il y a des limites aux contraintes, au nombre de colonnes, et ainsi de suite. Comparez la taille et le contenu de votre base de données aux spécifications de votre SGBD. Si vous vous rapprochez du point de rupture, il va vous falloir envisager une mise à niveau vers un système plus puissant. Une autre solution pourrait consister à archiver les données les plus anciennes (celles qui ne sont plus exploitées, celles qui servent uniquement à conserver la mémoire de vos activités et à établir des statistiques), puis à les supprimer de votre base.

Contraintes

Ci-avant dans ce chapitre, j'ai parlé des contraintes comme de mécanismes qui permettent de vous assurer que les données saisies dans une colonne d'une table appartiennent bien à la plage de valeurs autorisées. Une *contrainte* est une règle que le SGBD applique. Une fois la base de données définie, vous pouvez inclure des contraintes (telles que `NOT NULL`) dans la définition des tables. Il est alors de la responsabilité du SGBD d'interdire toute transaction qui violerait une contrainte.

Il existe trois types de contraintes :

- Une **contrainte de colonne** impose une condition à une colonne dans une table.
- Une **contrainte de tables** applique à une table tout entière.

> » Une **assertion** est une contrainte qui porte sur plusieurs tables.

Contraintes de colonne

Voici un exemple de contrainte de colonne dans une instruction DDL :

```
CREATE TABLE CLIENTS(
NOM_CLIENT CHARACTER (30) NOT NULL,
ADRESSE_1 CHARACTER (30),
ADRESSE_2 CHARACTER (30),
VILLE CHARACTER (25),
ETAT CHARACTER (2),
CODE_POSTAL CHARACTER (10),
TELEPHONE CHARACTER (13),

FAX CHARACTER (13),
CONTACT CHARACTER (30)
)  ;
```

L'instruction applique la contrainte `NOT NULL` à la colonne `NOM_CLIENT`, ce qui signifie que cette colonne ne peut contenir une valeur nulle. Vous pourriez aussi appliquer la contrainte UNIQUE. Elle spécifie que chaque valeur de cette colonne doit être unique dans toute la table. La contrainte `CHECK` est particulièrement utile parce qu'elle peut prendre n'importe quelle expression valide en argument. Par exemple :

```
CREATE TABLE TESTS (
NOM_TEST CHARACTER (30) NOT NULL,
PRIX_STANDARD NUMERIC (6,2)
CHECK (PRIX_STANDARD >= 0.0
AND PRIX_STANDARD <= 200.0)
)  ;
```

Le coût d'un test de VetLab doit toujours être supérieur ou égal à zéro. Et aucun test ne doit coûter plus de 200 euros. La clause `CHECK` refuse tout montant qui n'appartient pas à la plage `0 <= PRIX_STANDARD <= 200`. Vous pourriez aussi spécifier cette contrainte de la manière suivante :

```
CHECK (PRIX_STANDARD BETWEEN 0.0 AND 200.0)
```

Contraintes de table

La contrainte PRIMARY KEYspécifie que la colonne à laquelle elle s'applique est une clé primaire. Cette contrainte s'impose donc à la table entière et est l'équivalent d'une combinaison des contraintes de colonnes NOT NULL et UNIQUE. Vous pouvez spécifier cette contrainte dans une instruction CREATE, comme dans l'exemple suivant :

```
CREATE TABLE CLIENTS(
NOM_CLIENT CHARACTER (30)   PRIMARY KEY,
ADRESSE_1 CHARACTER (30),
ADRESSE_2 CHARACTER (30),
VILLE CHARACTER (25),
ETAT CHARACTER (2),
CODE_POSTAL CHARACTER (10),
TELEPHONE CHARACTER (13),
FAX CHARACTER (13),
CONTACT CHARACTER (30)
)  ;
```

Les contraintes nommées, telles que `NOM_CE` dans l'exemple donné dans « Les suppressions en cascade – à utiliser avec soin » peuvent avoir quelques fonctionnalités supplémentaires. Par exemple, supposons que vous souhaitiez charger quelques centaines de prospects dans votre table PROSPECT. Vous disposez d'un fichier qui contient essentiellement des prospects des États-Unis, mais on trouve aussi quelques prospects canadiens perdus dans le fichier. Normalement, vous souhaiteriez restreindre votre table PROSPECT pour n'y faire figurer que les prospects américains, mais vous ne voulez pas que le chargement soit interrompu

chaque fois qu'il tombe sur un canadien (les codes postaux canadiens comprennent des lettres et des chiffres, mais les codes postaux américains ne contiennent que des chiffres). Vous pouvez choisir de ne pas activer de contrainte sur CODE_POSTAL tant que le chargement n'est pas terminé, et d'activer la contrainte ensuite.

Au départ, votre table PROSPECT a été créée en utilisant cette commande CREATE TABLE :

```
CREATE TABLE PROPECT (
NOM_CLIENT CHARACTER (30)   PRIMARY KEY,
ADRESSE_1 CHARACTER (30),
ADRESSE_2 CHARACTER (30),
VILLE CHARACTER (25),
ETAT CHARACTER (2),
CODE_POSTAL CHARACTER (10),
TELEPHONE CHARACTER (13),
FAX CHARACTER (13),
CONTACT CHARACTER (30)
CONSTAINT ZIP CHECK (CODE_POSTAL BETWEEN 0 AND
99999)
)  ;
```

Avant le chargement, vous pouvez désactiver la contrainte ZIP :

```
ALTER TABLE PROSPECT
CONSTRAINT ZIP NOT ENFORCED;
```

Une fois que le chargement est terminé, vous pouvez réactiver la contrainte :

```
ALTER TABLE PROSPECT
CONSTRAINT ZIP ENFORCED;
```

À cet instant, vous pouvez éliminer les lignes qui ne répondent pas à la contrainte avec :

```
DELETE FROM PROSPECT
```

```
WHERE CODE_POSTAL NOT BETWEEN 0 AND 99999 ;
```

Les assertions

Une *assertion* spécifie une restriction portant sur plusieurs tables. L'exemple suivant utilise une condition de recherche qui extrait des valeurs de deux tables pour créer une assertion :

```
CREATE TABLE COMMANDES (
NUMERO_COMMANDE INTEGER NOT NULL,
NOM_CLIENT CHARACTER (30),
TEST_COMMANDE CHARACTER (30),
COMMERCIAL CHARACTER (30),
DATE_COMMANDE DATE ) ;

CREATE TABLE RESULTATS (
NUMERO_RESULTAT INTEGER NOT NULL,
NUMERO_COMMANDE INTEGER,
RESULTAT CHARACTER(50),
DATE_PRODUCTION DATE,
PRELIMINAIRE_FINAL CHARACTER (1) ) ;

CREATE ASSERTION
CHECK (NOT EXISTS SELECT * FROM COMMANDES,
RESULTATS
WHERE COMMANDE.NUMERO_COMMANDE =
RESULTATS.NUMERO_COM-
MANDE
AND COMMANDES.DATE_COMMANDE >
RESULTATS.DATE_PRODUCTION)
 ;
```

Cette assertion vous garantit qu'aucun résultat n'est transmis si la commande correspondante n'a pas été validée.

Normaliser la base de données

Certaines méthodes d'organisation des données sont meilleures que d'autres. Il y en a qui sont plus logiques. D'autres qui sont plus simples. Certaines permettent d'éviter l'apparition d'incohérences quand vous commencez à utiliser la base.

Les anomalies de modification et les formes normales

Nombre de problèmes (appelés *anomalies de modification*) sont capables de « pourrir » une base de données que vous n'aurez pas correctement structurée. Pour les éviter, vous pouvez normaliser la structure de la base de données. La normalisation consiste généralement à découper une table en deux tables plus simples.

Les *anomalies de modification* sont ainsi nommées parce qu'elles sont générées lors de l'ajout, de la modification ou de la suppression de données dans une table.

Pour comprendre comment elles peuvent survenir, observez la Figure 5.2.

Votre société commercialise des produits d'entretien ménagers et d'hygiène personnelle. Pour chaque produit, vous facturez le même prix à tous vos clients. La table VENTES conserve la trace de toutes les transactions. Supposez maintenant que la référence 1001 déménage et ne soit donc plus un de vos clients. Ce qu'il a acheté par le passé ne vous intéresse plus. Vous voulez retirer sa ligne de la table. Mais en effectuant cette suppression, vous n'allez pas seulement oublier que le client 1001 a acheté de la poudre à laver ; vous allez aussi « oublier » que cette poudre coûte 12 euros. Cette situation est appelée anomalie de suppression. En supprimant un fait (le client 1001 a acheté de la poudre à laver), vous allez en effacer un autre par inadvertance (le prix de la poudre en question).

VENTES

Client_ID	Produits	Prix
1001	Poudre à laver	12
1007	Dentifrice	3
1010	Détergent	4
1024	Dentifrice	3

FIGURE 5.2 La table VENTES.

La même table permet de montrer ce qu'est une anomalie d'insertion. Par exemple, supposons que vous désiriez ajouter un déodorant à 2 euros à votre inventaire. Vous ne pouvez pas ajouter cette donnée dans la table VENTES, car personne encore n'a acheté ce produit.

Le problème avec notre table VENTES est qu'elle traite de trop d'informations en même temps. Elle enregistre les produits que les clients achètent et le prix de ces produits. Vous devez donc scinder la table en deux, chaque nouvelle table s'occupant d'un thème ou d'une idée, comme sur la Figure 5.3.

La Figure 5.3 vous montre le résultat de la division de la table VENTES en deux nouvelles tables :

- ACHAT_CLIENT traite de la seule idée d'un achat d'un client.
- PRIX_PRODUIT traite de la seule idée du prix d'un produit.

ACHAT_CLIENT

Client_ID	Produits
1001	Poudre à laver
1007	Dentifrice
1010	Détergent
1024	Dentifrice

PRIX_PRODUIT

Produits	Prix
Poudre à laver	12
Dentifrice	3
Détergent	4

FIGURE 5.3 La table VENTES est découpée en deux tables.

Vous pouvez maintenant supprimer la ligne qui concerne le client 1001 de ACHAT_CLIENT sans perdre le prix de la poudre à laver. Cette information se trouve maintenant dans la table PRIX_PRODUIT. Il devient aussi possible d'ajouter le déodorant à PRIX_PRODUIT qu'il ait déjà été vendu ou non. Les informations relatives aux achats sont stockées ailleurs, dans ACHAT_CLIENT.

Le processus qui consiste à découper une table en plusieurs tables dont chacune traite d'un thème particulier est appelé normalisation. Un processus de normalisation qui résout un problème ne règle pas forcément les autres. Pour les résoudre tous, il vous faudra probablement découper encore les tables jusqu'à ce que chacune ne s'occupe plus que d'une seule idée. Chaque table devrait donc prendre en charge un et un seul thème principal. Parfois, il est véritablement difficile de déterminer si une table traite d'un ou de plusieurs thèmes.

Vous pouvez classer les tables en fonction du type d'anomalies de modification auquel elles sont sujettes. Dans un document de 1970 qui décrit pour la première fois le modèle relationnel, E.F. Codd identifie trois sources d'anomalies de modification et définit la première, la deuxième et la troisième forme normale (1NF, 2NF et 3NF) pour les prévenir. Dans les années suivantes, Codd et d'autres chercheurs ont découvert d'autres types d'anomalies et ont spécifié de nouvelles formes normales pour les traiter. La forme normale de Boyce-Codd (BCNF), la quatrième forme normale (4NF) et la cinquième forme normale (5NF) offrent chacune un degré supérieur de protection contre les anomalies de modification. Cependant, ce n'est qu'en 1981 qu'un article publié par R. Fagin décrivit la forme normale domaine/clé (DK/NF). Cette dernière forme normale permet de garantir qu'une table sera toujours exempte d'anomalies de modification.

Les formes normales sont imbriquées dans le sens où une table vérifiant la deuxième forme normale (2NF) répond obligatoirement à la première (1NF). De même, une table au niveau 3NF est automatiquement conforme à 2NF, et ainsi de suite. Dans la plupart des applications pratiques, la troisième forme normale est suffisante pour assurer un haut degré d'intégrité à la base de données. Pour avoir une certitude absolue, la forme DK/NF s'impose.

Une fois que vous aurez normalisé une base autant qu'il est possible, vous pourrez procéder à des dénormalisations sélectives pour améliorer les performances. Faites alors attention aux types d'anomalies qui pourraient survenir.

Première forme normale

Pour respecter la première forme normale, une table doit satisfaire aux conditions suivantes :

- La table a deux dimensions, avec des lignes et des colonnes.
- Chaque ligne contient des données qui se rapportent à une chose ou à une portion d'une chose.
- Chaque colonne contient des données qui correspondent à un seul attribut de la chose décrite.
- Chaque cellule (intersection d'une ligne et d'une colonne) ne doit contenir qu'une seule valeur.
- Les entrées d'une colonne doivent toutes être du même type. Par exemple, si l'entrée d'une ligne de la colonne contient le nom d'un employé, toutes les autres lignes de la colonne doivent contenir le nom d'un employé.
- Chaque colonne doit porter un nom unique.
- Deux lignes ne peuvent être identiques (et donc chaque ligne doit être unique).
- L'ordre des colonnes et l'ordre des lignes n'ont aucune signification.

Une table (ou relation) conforme à la première forme normale est immunisée contre certaines anomalies de modification mais peut être sujette à d'autres. La table VENTES de la Figure 5.2 est en première forme

normale et souffre d'anomalies de suppression et d'insertion. Par conséquent, la première forme normale peut suffire à certaines applications, mais pas à d'autres.

Deuxième forme normale

Pour apprécier ce qu'est la deuxième forme normale, vous devez appréhender le concept de dépendance fonctionnelle. Une dépendance fonctionnelle est une relation entre attributs. Un attribut est fonctionnellement dépendant d'un autre si la valeur du second détermine celle du premier. Si vous connaissez la valeur du second attribut, vous pouvez donc déterminer celle du premier.

Par exemple, supposons qu'une table dispose des attributs (colonnes) `PRIX_STANDARD`, `NOMBRE_TESTS` et `PRIX_STANDARD_TOTAL` et qu'ils sont utilisés dans l'équation suivante :

```
PRIX_STANDARD_TOTAL = PRIX_STANDARD *
NOMBRE_TESTS
```

`PRIX_STANDARD_TOTAL` dépend fonctionnellement de `PRIX_STANDARD` et de `NOMBRE_TESTS`. Si vous connaissez les valeurs de `PRIX_STANDARD` et de `NOMBRE_TESTS`, vous pouvez calculer `PRIX_STANDARD_TOTAL`.

Toute table qui se trouve dans la première forme normale doit avoir une clé primaire unique. Cette clé peut être formée à partir d'une ou de plusieurs colonnes. Une clé qui porte sur plus d'une colonne est dite clé composite. Pour vérifier la deuxième forme normale (2NF), toutes les colonnes qui ne servent pas à composer la clé doivent dépendre de cette dernière. Ainsi, toute relation en 1NF, et dont la clé correspond à un unique attribut, est automatiquement en 2NF. Si une relation est une clé composite, tous les autres attributs doivent dépendre de ceux qui composent cette clé.

En d'autres termes, si vous disposez d'une table dans laquelle certains attributs ne participant pas à la clé ne dépendent pas de tous les composants de ladite clé, découpez cette table en deux parties ou plus, et ce jusqu'à ce que tous les attributs « non clé » de chaque table dépendent de ceux qui en composent la clé primaire.

À ce stade, un exemple permettra d'y voir plus clair. Reprenons la table VENTES de la Figure 5.2. Mais plutôt que d'enregistrer une seule commande par client, vous ajoutez une ligne chaque fois que le client achète un article pour la première fois. De plus, certains clients (dont CLIENT_ID est compris entre 1001 et 1007) ont droit à une réduction particulière. La Figure 5.4 reprend quelques lignes de cette table.

Sur la Figure 5.4, `CLIENT_ID` n'identifie pas de manière unique une ligne (deux lignes comportent la valeur 1001, et dans deux autres lignes il prend la valeur 1010). Cependant, la combinaison de `CLIENT_ID` et de `PRODUIT` identifie de manière unique une ligne. Les deux colonnes forment une clé composite.

VENTES

CLIENT_ID	Produits	Prix
1001	Poudre à laver	11.00
1007	Dentifrice	2.70
1010	Détergent	4.00
1024	Dentifrice	3.00
1010	Poudre à laver	12.00
1001	Dentifrice	2.70

FIGURE 5.4 Dans la table VENTES, les colonnes CLIENT_ID et PRODUIT constituent une clé composite.

Sans le fait que certains clients obtiennent une réduction particulière, la table ne serait pas en deuxième forme normale. En effet, `PRIX` (un attribut qui ne sert pas à former la clé) ne dépendrait que d'une partie de la clé composite (`PRODUIT`). Du fait que certains clients ont droit à une réduction, `PRIX` peut varier en fonction des valeurs des colonnes `CLIENT_ID` et `PRODUIT`. La table vérifie donc la deuxième forme normale.

Troisième forme normale

Les tables de la deuxième forme normale sont toujours sujettes à certains types d'anomalies de modification. Ces anomalies sont engendrées par des dépendances transitives.

Une dépendance transitive se produit quand un attribut dépend d'un deuxième attribut qui dépend à son tour d'un troisième. Supprimer des lignes dans une table qui possède ce genre de dépendance peut entraîner une perte d'informations. Une relation qui se trouve en troisième forme normale est une relation qui se trouve en deuxième forme normale et ne contient aucune dépendance transitive.

Reprenons la table VENTES de la Figure 5.2. Vous savez déjà qu'elle est en première forme normale. Tant que vous n'autorisez qu'une ligne par `CLIENT_ID`, votre clé primaire n'a besoin que d'un seul attribut, si bien que la table est en deuxième forme normale. Cependant, la table est toujours sujette à des anomalies.

Que se passe-t-il si le 1010 n'est pas satisfait de son détergent et qu'il demande à être remboursé ? Vous devrez supprimer la troisième ligne de la table. Vous avez maintenant un problème, car vous allez aussi perdre l'information qui vous indique que le détergent vaut 4 euros. C'est un exemple de dépendance transitive : `PRIX` dépend de `PRODUIT`, qui lui-même dépend de la clé primaire `CLIENT_ID`.

Il vous faut découper la table VENTES en deux tables pour résoudre le problème de la dépendance transitive. Sur la Figure 5.3, les deux tables ACHAT_CLIENT et PRIX_PRODUIT forment une base de données qui vérifie la troisième forme normale.

La forme normale domaine/clé (DKNF)

Une fois que vous avez passé une base de données en troisième forme normale, vous avez éliminé presque toute possibilité de voir survenir une anomalie de modification (mais pas encore *toutes* les possibilités). Les formes normales au-delà de la troisième sont définies pour éliminer les quelques risques restants. La forme normale Boyce-Codd (BCNF), la quatrième forme normale (4NF) et la cinquième forme normale (5NF) sont

quelques-unes de ces formes supplémentaires. Chacune prévient l'apparition d'anomalies de modification dans un cas particulier, mais pas dans tous. Cela, seule la forme normale domaine/clé (DKNF) vous le garantit.

Une relation est dans une *forme normale domaine/clé* si chaque contrainte de la relation est une conséquence logique de la définition des clés et des domaines. Dans ce schéma, une contrainte correspond à toute règle assez clairement énoncée pour que vous puissiez vérifier qu'elle s'applique ou non (en d'autres termes, si elle peut être évaluée comme étant vraie ou fausse). Une *clé* est un identifiant unique pour les lignes de la table. Un *domaine* est un ensemble de valeurs autorisées pour un attribut.

Considérez encore une fois la base de données de la Figure 5.2, qui est en 1NF. Que devriez-vous faire pour la passer en DKNF ?

Table : VENTES (CLIENT_ID, PRODUIT, PRIX)

Clé : CLIENT_ID

Contraintes : 1. CLIENT_ID détermine PRODUIT

2. PRODUIT détermine PRIX

3. CLIENT_ID doit être un entier > 1 000

Pour renforcer la troisième contrainte, vous pouvez modifier la définition du domaine de CLIENT_ID afin d'y inclure cette contrainte. Cette dernière est alors la conséquence logique de la définition du domaine de la colonne CLIENT_ID. PRODUIT dépend de CLIENT_ID et CLIENT_ID est une clé, si bien que la contrainte 1 est vérifiée, et est de surcroît une conséquence logique de la définition de la clé. La contrainte 2 pose un problème. PRIX dépend de (est une conséquence logique de) PRODUIT qui n'est pas une clé. La solution consiste à diviser la table VENTES en deux tables. L'une de ces tables utilise CLIENT_ID pour clé tandis que l'autre prend PRODUIT pour clé. C'est la configuration de la Figure 5.3. La base de données qui y est représentée, en plus d'être en 3NF, est aussi conforme aux spécifications DKNF.

Concevez vos bases de données pour qu'elles soient si possible en DKNF. Il ne pourra alors plus survenir d'anomalies de modification. Si la structure d'une base de données vous interdit de vous conformer à DKNF, vous devrez intégrer des contrôles de contraintes dans l'application qui utilise la base de données. En effet, la base de données ne garantit pas par elle-même

que ces contraintes seront respectées.

La forme anormale

Adopter une forme anormale est parfois une bonne solution. En effet, si vous poussez la normalisation trop loin, vous risquez de générer tellement de tables que la base de données en deviendra totalement ingérable et inefficace. De plus, les performances pourraient s'effondrer au passage. Une structure viable est souvent un peu dénormalisée. En pratique, les bases de données ne sont presque jamais totalement en DKNF. Pour autant, vous devez tenter de normaliser le plus possible les bases de données que vous concevez afin d'éliminer au maximum les risques de corruption des données résultant d'anomalies de modification.

Une fois la base de données normalisée autant que faire se peut, revenez en arrière. Si les performances ne sont pas satisfaisantes, réexaminez votre travail de conception et voyez si une dénormalisation sélective serait à même d'améliorer ces performances sans sacrifier l'intégrité. En ajoutant avec grand soin une certaine dose de redondance dans des emplacements stratégiques, et en acceptant une part de dénormalisation, vous pourrez arriver à une base de données qui sera à la fois efficace et à l'abri des anomalies.

Enregistrer et extraire des données

DANS CETTE PARTIE...

Gérer des données.

Suivre le temps.

Traiter des valeurs.

Construire des requêtes.

Visiter le contenu en ligne des Nuls sur www.dummies.com/extras/sql.

Chapitre 6
Manipuler les données d'une base

DANS CE CHAPITRE :

- **Utiliser des données.**
- **Extraire d'une table les données dont vous avez besoin.**
- **N'afficher que les informations voulues à partir d'une ou plusieurs tables.**
- **Mettre à jour des informations dans des tables et des vues.**
- **Ajouter une nouvelle ligne à une table.**
- **Modifier toutes ou certaines des données d'une ligne.**
- **Supprimer une ligne dans une table.**

Les Chapitres 3 et 4 vous ont appris combien il est indispensable de donner à votre base une bonne structure pour préserver l'intégrité des données qu'elle contient. Mais plus que la structure, ce sont les données qui vous intéressent. Vous voulez pouvoir les ajouter à des tables, les extraire et les afficher, les modifier, et enfin les supprimer.

En principe, la manipulation des données contenues dans une base est assez simple. Il n'est pas compliqué de comprendre comment ajouter des données à une table – vous pouvez le faire ligne par ligne ou en une seule fois. Modifier, supprimer ou extraire des lignes d'une table n'a rien de difficile en pratique. En fait, la véritable difficulté est d'arriver à *sélectionner* les lignes que vous voulez modifier, extraire ou supprimer. Extraire des informations peut se révéler aussi compliqué que de rechercher une aiguille dans une meule de foin, car les données que vous recherchez peuvent être noyées dans une base extrêmement volumineuse. Fort heureusement, et pour peu que vous sachiez formuler votre recherche, l'instruction SQL SELECT, l'ordinateur fera tout le travail à votre place. D'accord. Dire qu'il est facile

de manipuler une base de données avec SQL, d'ajouter, de modifier, de supprimer ou de retrouver des informations est peut-être un peu exagéré...

Commençons donc par un exemple *réellement* simple.

Extraire des données

La manipulation des données à laquelle les utilisateurs procèdent le plus souvent est l'extraction d'informations sélectionnées dans une base. Vous voulez par exemple récupérer le contenu d'une certaine ligne parmi les milliers que comporte une table. Ou bien vous voulez trouver toutes les lignes qui satisfont un critère ou une combinaison de critères. Il se peut aussi que vous ayez besoin de récupérer toutes les lignes d'une table. L'instruction `SELECT` effectue toutes ces tâches à votre place.

La forme la plus simple de l'instruction `SELECT` est celle qui retourne toutes les lignes d'une table :

```
SELECT * FROM CLIENTS ;
```

L'étoile (*) est un caractère de substitution (un joker) qui désigne tout. Dans ce contexte, l'étoile correspond à tous les noms de colonnes de la table `CLIENTS`. Il en résulte que toutes les lignes de toutes les colonnes de cette table vous sont retournées. En d'autres termes, l'instruction ci-dessus renvoie la totalité de la table.

Les instructions `SELECT` peuvent être bien plus complexes que celle présentée dans l'exemple précédent. En fait, certaines instructions `SELECT` sont si complexes qu'elles en sont virtuellement indéchiffrables. Cette complexité potentielle tient au fait que vous pouvez accoler de nombreuses clauses modificatrices à l'instruction. Le Chapitre 10 présente ces clauses dans le détail. Je vais ici brièvement parler de la clause `WHERE`, qui est la plus communément employée pour limiter le nombre de lignes retournées par une instruction `SELECT`.

Une instruction SELECT accompagnée de la clause `WHERE` adopte la forme suivante :

```
SELECT liste_colonnes FROM nom_table
```

```
    WHERE condition ;
```

`Liste_colonnes` spécifie quelles colonnes vous voulez dans la table. L'instruction n'affichera que ces colonnes. La clause `FROM` indique la table d'où vous voulez extraire ces colonnes. La clause `WHERE` exclut toutes les lignes qui ne répondent pas à vos conditions. Une condition peut être simple (par exemple, `WHERE ETAT_CLIENT = 'CA'`) ou composée (par exemple `WHERE ETAT_CLIENT = 'CA' AND STATUT = 'Actif'`). L'exemple suivant montre à quoi peut ressembler une requête qui comprend une condition composite :

```
SELECT NOM_CLIENT, TELEPHONE_CLIENT FROM CLIENTS
    WHERE ETAT_CLIENT = 'CA'
    AND STATUT = 'Actif' ;
```

SQL ET LES OUTILS PROPRIÉTAIRES

Les instructions `SELECT` ne constituent pas le seul moyen d'extraire des données d'une base. Si vous utilisez un SGBD, ce dernier dispose probablement d'outils propriétaires conçus pour cet usage. Vous pouvez utiliser ces outils (qui sont généralement assez intuitifs) pour ajouter, supprimer, modifier et extraire des données.

De nombreux front-end de SGBD vous permettent de choisir entre utiliser ces outils propriétaires ou utiliser SQL. Il arrive parfois que les outils propriétaires ne permettent pas d'obtenir tout ce qu'il est possible d'exprimer en SQL. Si vous devez effectuer une opération que ces outils ne supportent pas, il vous faudra utiliser SQL. Il est donc particulièrement utile de se familiariser avec ce langage, même si vous comptez utiliser un outil propriétaire la plupart du temps. Pour accomplir avec succès une opération trop complexe pour votre outil propriétaire, il faut que vous ayez les idées au clair sur le fonctionnement de SQL et sur ce qu'il est capable de faire.

L'instruction retourne les noms et numéros de téléphone de tous les clients actifs qui vivent en Californie. Le mot clé `AND` signifie que vous ne voulez récupérer que les lignes qui vérifient les deux conditions que vous posez.

Créer des vues

La structure d'une base conçue en respectant les règles du genre (dont notamment une normalisation appropriée, voir Chapitre 5) optimise l'intégrité des données. Pour autant, cette structure n'est souvent pas la plus adaptée pour représenter visuellement les données qu'elle contient. Plusieurs applications peuvent utiliser les mêmes données, mais chacune les affiche à sa manière. L'une des fonctionnalités les plus puissantes de SQL est la possibilité de générer des représentations des données – les vues – dont la structure diffère de celle de la base. Les tables dont vous utilisez les lignes et les colonnes dans une vue sont les tables de base. Le Chapitre 3 traite des vues qui sont un des éléments du langage de définition des données (DDL). Dans cette section, j'explique à quoi servent les vues dans le contexte de la récupération et de la manipulation des données.

Une instruction `SELECT` retourne toujours un résultat sous la forme d'une table virtuelle. Une vue est une variété de table virtuelle. Vous pouvez différencier une vue des autres variétés de tables virtuelles, car sa définition est mémorisée dans les métadonnées de la base, ce qui lui confère une persistance que n'ont pas les autres tables virtuelles.

Vous pouvez manipuler une vue comme s'il s'agissait d'une table normale. La différence est que les données d'une vue n'ont pas d'existence indépendante. Elles proviennent en effet d'une ou plusieurs tables (celles qui ont justement servi à définir la vue). Chaque application est donc à même d'appliquer *son propre point de vue* sur un ensemble de données.

Reprenons l'exemple de la base de données décrite au Chapitre 5. Cette base contient cinq tables : `CLIENTS`, `TESTS`, `EMPLOYES`, `COMMANDES` et `RESULTATS`. Supposons que le responsable du marketing pour les États-Unis désire connaître les États d'où proviennent les commandes. Une partie de cette information se trouve dans la table `CLIENTS` et une autre dans la table `COMMANDES`. Supposons maintenant que le responsable du contrôle qualité désire comparer la date à laquelle un test est commandé et celle à

laquelle le résultat final est produit. Cette comparaison requiert des données des tables COMMANDES et RESULTATS. Pour répondre aux besoins de ces deux personnes, vous pouvez créer des vues qui contiennent les données dont vous avez besoin dans chaque cas.

À partir de tables

Pour le responsable du marketing, vous pouvez créer la vue représentée Figure 6.1.

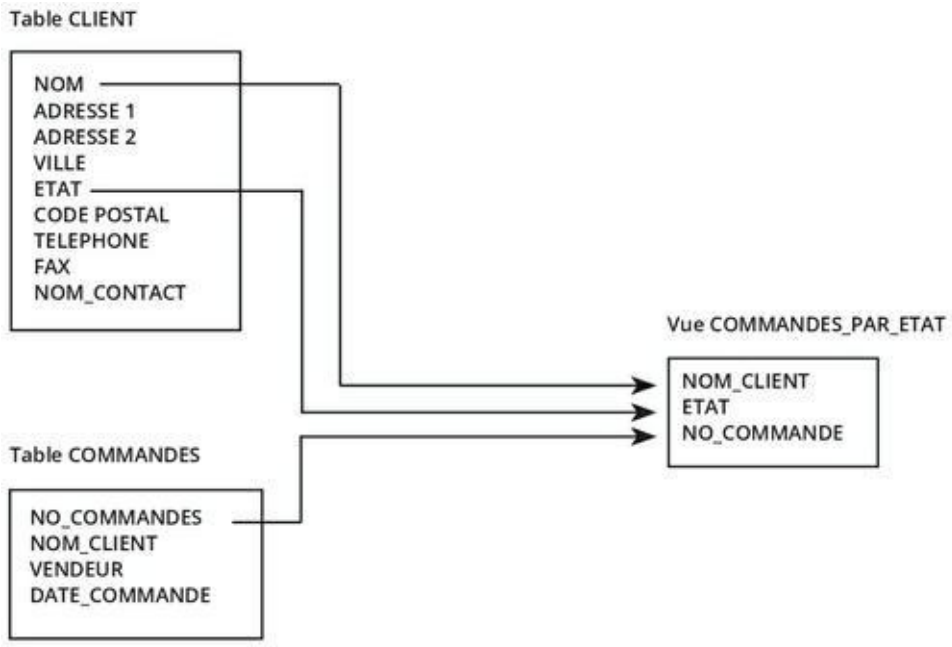


FIGURE 6.1 La vue COM-MANDES_ PAR_ETAT pour le responsable marketing.

L'instruction suivante crée la vue souhaitée :

```
CREATE VIEW COMMANDES_PAR_ETAT
      (NOM_CLIENT, ETAT, NUMERO_COMMANDE)
    AS SELECT CLIENTS.NOM_CLIENT, ETAT,
NUMERO_COMMANDE
    FROM CLIENTS, COMMANDES
    WHERE CLIENTS.NOM_CLIENT =
COMMANDES.NOM_CLIENT ;
```

La nouvelle vue dispose de trois colonnes, `NOM_CLIENT`, `ETAT` et `NUMERO_COMMANDE`. `NOM_CLIENT` apparaît à la fois dans `CLIENTS` et dans `COMMANDES`, et il sert de lien entre ces deux tables. La vue tire l'information `ETAT` de la table `CLIENTS` et l'information `NUMERO_COMMANDE` de la table `COMMANDES`.

Dans l'exemple précédent, vous déclarez explicitement le nom des colonnes de la nouvelle vue. Cette précision n'est pas nécessaire si les noms des colonnes de la vue sont les mêmes que ceux des colonnes auxquelles elles correspondent dans les tables sources. L'exemple de la section suivante présente l'utilisation d'une instruction `CREATE VIEW` où les noms des colonnes sont fournis de manière implicite.

Avec un critère de sélection

Le responsable qualité a besoin d'une vue différente de la précédente. Elle est décrite Figure 6.2.

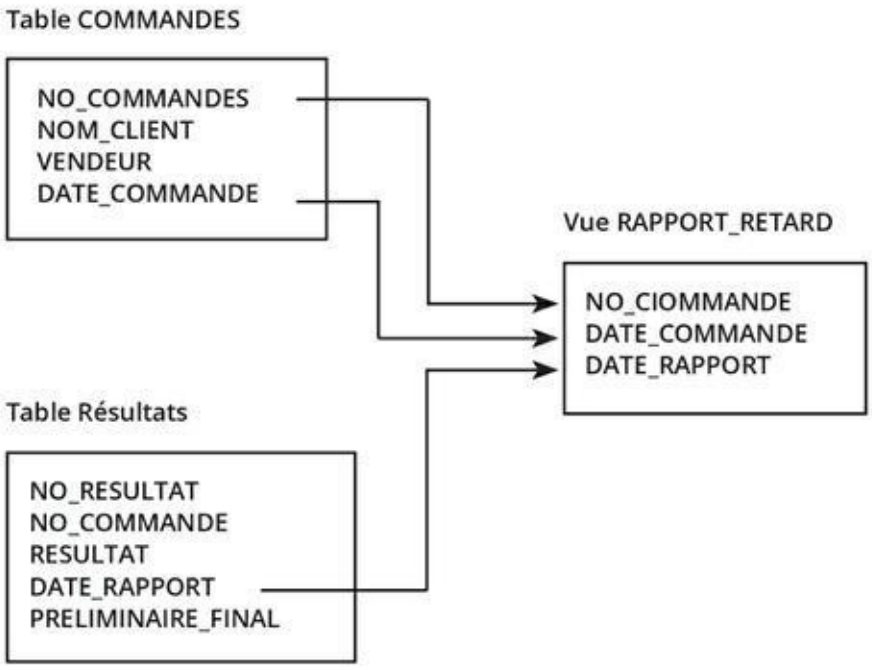


FIGURE 6.2 La vue RAPPORT_RE-TARD pour le responsable qualité.

L'instruction suivante crée la vue :

```
CREATE VIEW RAPPORT_RETARD
    AS SELECT COMMANDES.NUMERO_COMMANDE,
DATE_COMMANDE,
DATE_RAPPORT
    FROM COMMANDES, RESULTATS
    WHERE COMMANDES.NUMERO_COMMANDE =
RESULTATS.NUME-
RO_COMMANDE
    AND RESULTATS.PRELIMINAIRE_FINAL = 'F' ;
```

Cette vue contient l'information DATE_COMMANDE en provenance de la table COMMANDES, et les dates des résultats définitifs fournis par la table RESULTATS. Seules les lignes de la table RESULTATS dont la colonne PRELIMINAIRE_FINAL contient la valeur « F » apparaissent dans la vue. Notez en passant que la liste des colonnes spécifiée dans la vue COMMANDES_PAR_ETAT est facultative. La vue fonctionnerait parfaitement sans cette liste.

Avec un attribut modifié

La clause SELECT des deux exemples précédents ne contient que des noms de colonnes. Cependant, vous pouvez aussi y inclure des expressions. Supposons que le propriétaire de VetLab fête l'anniversaire de sa société et désire le célébrer en offrant une réduction de 10 % à tous ses clients. Il peut créer une vue basée sur les tables TESTS et COMMANDES en utilisant l'instruction suivante :

```
CREATE VUE_ANNIVERSAIRE
(NOM_CLIENT, TEST, DATE_COMMANDE,
PRIX_ANNIVERSAIRE)
AS SELECT NOM_CLIENT, TEST_COMMANDE,
DATE_COMMANDE,
PRIX_STANDARD * .9
FROM COMMANDES, TESTS
```

```
WHERE TEST_COMMANDE = NOM_TEST ;
```

La Figure 6.3 montre comment créer cette vue.

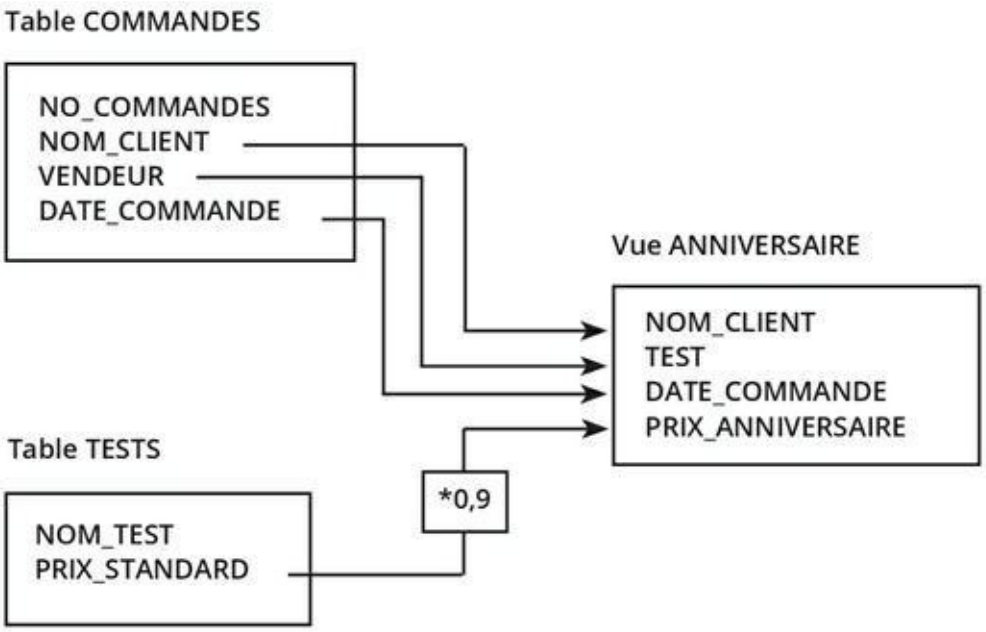


FIGURE 6.3 La vue créée pour afficher les prix anniversaire.

Vous pouvez créer une vue à partir de plusieurs tables, comme dans les exemples précédents, ou à partir d'une seule table. Si vous n'avez pas besoin de certaines colonnes ou lignes d'une table, créez une vue qui les élimine. Cela vous évitera de modifier accidentellement le contenu d'une colonne sur laquelle vous ne souhaitez pas travailler.

Une autre bonne raison de créer des vues est d'assurer la sécurité des tables sous-jacentes. Si vous voulez permettre la consultation de quelques colonnes d'une table tout en dissimulant les autres, vous pouvez créer une vue ne contenant que les colonnes que vous souhaitez rendre publiques. Vous n'avez plus alors qu'à autoriser largement l'accès à la vue tout en restreignant fortement l'accès à la table sous-jacente. C'est un bon procédé pour la sécuriser. Le Chapitre 14 explore la sécurité des bases de données et explique comment accorder et révoquer des privilèges.

Modifier les vues

À peine créée, une table est en soi susceptible d'être remplie, éditée, modifiée, vidée... Les vues n'offrent pas nécessairement les mêmes

possibilités. Si vous éditez une vue, vous modifiez en réalité la table sous-jacente. Voici quelques problèmes potentiels que vous pouvez rencontrer lorsque vous actualisez des vues :

- **Certaines vues prennent leurs données dans plusieurs tables.** Si vous modifiez une telle vue, comment savoir laquelle des tables sous-jacentes sera modifiée ?
- **Une vue peut inclure une expression dans une liste** `SELECT`. Du fait que les expressions ne correspondent pas directement à des lignes de vos tables, le SGBD ne sait pas comment les mettre à jour.

Supposons que vous ayez créé une vue en utilisant l'instruction suivante :

```
CREATE VIEW COMP (NomEmploye, Paye)
    AS SELECT NomEmploye, Salaire+Comm AS Paye
    FROM EMPLOYES ;
```

Pouvez-vous mettre à jour Paye en utilisant l'instruction suivante ?

```
UPDATE COMP SET Paye = Paye + 100 ;
```

Cela n'aurait aucun sens car la table sous-jacente ne contient pas de colonne `Paye`. Vous ne pouvez pas mettre à jour quelque chose qui n'existe pas dans la table sous-jacente.

Souvenez-vous de ceci chaque fois que vous utilisez une vue : vous ne pouvez modifier une colonne de la vue que si elle correspond à une colonne d'une table de base sous-jacente.

Ajouter de nouvelles données

Une table qui vient d'être créée (que ce soit à l'aide d'une instruction SQL ou via un RAD) est un ensemble structuré qui ne contient aucune information. Une coquille vide, donc. Pour la rendre utile, il faut y stocker des données, dont vous disposez peut-être déjà (ou non) sous une forme

numérique. Ces données peuvent apparaître sous l'une des formes suivantes :

- **Elles ne sont pas encore compilées sous une forme numérique quelconque**. Quelqu'un devra probablement utiliser son clavier et son moniteur pour les saisir manuellement, une par une. Vous pourriez aussi utiliser un scanner ou un système de reconnaissance vocale, mais ce type de saisie est encore peu répandu (et surtout d'une fiabilité insuffisante).
- **Elles existent déjà sous une forme numérique** mais dans un format différent de celui utilisé par les tables où vous comptez les stocker. Vous devrez tout d'abord les convertir dans le format approprié puis les insérer dans la base.
- **Elles sont déjà compilées dans le bon format numérique**. Vous êtes prêt à opérer le transfert vers une nouvelle base de données.

Les sections qui suivent décrivent les procédures d'insertion des données dans une table en fonction de la forme qu'elles prennent. Selon le cas, vous pourrez importer toutes vos données en une seule fois ou bien enregistrement par enregistrement. Chaque enregistrement correspond à une ligne de la table.

Ajouter une ligne à la fois

La plupart des SGBD permettent de saisir des données via un formulaire. Cela vous permet de créer des écrans qui contiennent des champs pour chaque colonne d'une table. Le libellé des champs de saisie dans le formulaire renseigne l'utilisateur sur la colonne correspondante (si c'est écrit Date de commande, tout le monde est censé comprendre qu'il faut entrer ici la date de la commande en cours de traitement). L'opérateur saisit dans le formulaire toutes les données à enregistrer dans une seule et même

ligne. Une fois que le SGBD a accepté cette nouvelle ligne, le système rafraîchit le formulaire pour permettre la saisie d'une nouvelle série de données. De cette manière, il est simple (quoique fastidieux) de remplir la table ligne par ligne.

Un système de formulaires est facile à comprendre et moins sujet aux erreurs de saisie que l'utilisation par exemple de listes *de valeurs séparées par des virgules (format cs*v). Le principal problème, cependant, est qu'il n'existe aucun standard en ce domaine. Chaque SGBD propose sa propre solution pour créer des formulaires. Pour autant, cette diversité ne pose pas de problème à l'opérateur, car les formulaires adoptent presque toujours le même aspect d'un SGBD à un autre. C'est le développeur de l'application qui doit reprendre sa formation à zéro chaque fois qu'il change de SGBD, pas l'opérateur. Un autre problème possible tient au fait que certaines implémentations ne permettent pas de mettre en œuvre dans les formulaires tous les contrôles de validation que vous souhaiteriez effectuer lors de la saisie des données.

La meilleure solution pour maintenir l'intégrité des données de la base consiste en premier lieu à limiter la saisie de données erronées. Pour cela, vous devez appliquer des contraintes sur les champs de saisie d'un formulaire. Vous serez ainsi certain que la base de données n'accepte que des valeurs du bon type et qui appartiennent aux bonnes plages de valeurs. Bien entendu, ces contraintes ne peuvent pas permettre de prévenir toutes les erreurs possibles et imaginables.

Si l'outil de conception de formulaires de votre SGBD ne vous permet pas d'appliquer tous les contrôles requis pour préserver l'intégrité des données, vous pouvez créer votre propre écran, stocker les données saisies dans des variables, puis tester ces variables en utilisant du code applicatif. Une fois que vous serez certain que les données saisies sont valides, vous pourrez insérer la ligne en faisant appel à la commande SQL `INSERT`.

Utilisez la commande SQL `INSERT` pour insérer les données dans une ligne d'une table :

```
INSERT INTO table_1 [(colonne_1, colonne_2, ...,
co-
lonne_n)]
    VALUES (valeur_1, valeur_2, ..., valeur_n) ;
```

Comme le laissent entendre les crochets ([]), la liste des noms de colonnes est optionnelle. La liste par défaut correspond à l'ensemble des colonnes de la table (dans l'ordre où elles sont définies dans celle-ci). Si vous fournissez les valeurs dans l'ordre des colonnes de la table, ces éléments seront enregistrés au bon endroit. Par contre, si les valeurs sont placées dans un ordre différent, vous devez explicitement indiquer quelle colonne doit prendre quelle valeur.

Utilisez par exemple la syntaxe suivante pour saisir un enregistrement dans la table CLIENTS :

```
INSERT INTO CLIENTS (CLIENT_ID, PRENOM, RUE,
VILLE, ETAT,
CODE_POSTAL,
    TELEPHONE)
    VALUES (:vclientid, 'David', 'Taylor', '235
Nutley
Ave.',
    'Nutley', 'NJ', '07110', '(201) 555-1963') ;
```

Le premier élément de la liste `VALUE`, , `vclientid`, est une variable que le code de votre programme incrémente chaque fois que vous ajoutez une nouvelle ligne dans la table. De la sorte, vous êtes certain que deux lignes ne contiendront pas la même valeur `CLIENT_ID`. Cette colonne est en effet la clé primaire pour cette table, et chaque identifiant doit rester unique. Les valeurs restantes sont de simples données (plutôt que des variables *contenant* ces données). Bien entendu, vous pourriez, si vous le souhaitiez, stocker toutes ces données dans des variables. Les arguments du mot clé `VALUES` peuvent en effet être aussi bien des variables que des copies explicites des données elles-mêmes.

Ajouter des données à des colonnes choisies

Vous voulez parfois noter l'existence d'un objet sans pour autant disposer de toutes les informations qui le concernent. Vous pouvez alors insérer une ligne pour cet objet sans renseigner toutes les colonnes de la table. Pour

vous assurer que la table soit en première forme normale, cependant, vous devrez fournir suffisamment de données pour que toutes les lignes restent distinctes (pour plus d'informations sur la première forme normale, voir le Chapitre 5). Spécifier la clé primaire de la nouvelle ligne suffit à satisfaire cette condition. Les colonnes que vous ne renseignerez pas contiendront une valeur nulle.

Voici un exemple de saisie partielle :

```
INSERT INTO CLIENTS (CLIENT_ID, PRENOM, NOM)
    VALUES (:vclientid, 'Tyson', 'Taylor') ;
```

Vous n'insérez que le numéro d'identification unique du client, son nom et son prénom.

Ajouter un bloc de lignes à une table

Alimenter une base de données ligne par ligne en utilisant des instructions `INSERT` est particulièrement astreignant (et même carrément roboratif), surtout si vous devez le faire toute la journée. Même la plus élaborée des interfaces graphiques ne vous évitera pas de sombrer dans l'ennui. Clairement, il est toujours plus agréable de disposer d'une solution fiable permettant d'automatiser la saisie des données que d'obliger quelqu'un à rester assis toute la journée devant son clavier et ses listings de données.

Automatiser la saisie est possible quand, par exemple, les données existent déjà sous une forme numérique (quelqu'un les a déjà saisies manuellement). Inutile de faire bégayer l'histoire. Un ordinateur est tout à fait capable de transférer des données d'un fichier à un autre avec un minimum d'intervention humaine. Pour peu que vous connaissiez les caractéristiques de la donnée source et le format de la table destination, vous pourrez (en principe) automatiser le transfert.

Copier depuis un fichier de données étranger

Supposons que vous soyez en train de créer une base de données pour une nouvelle application. Certaines des données dont vous avez besoin existent déjà dans un fichier informatique. Il peut s'agir d'un fichier plein texte ou bien d'une table dans une base que gère un autre SGBD. Ces données sont peut-être en ASCII ou en EBCDIC, ou dans n'importe quel format propriétaire. Comment procéder ?

Priez tout d'abord pour que les données soient dans un format largement utilisé. Si c'est le cas, il se peut qu'il existe un outil de conversion capable d'en réaliser la traduction dans un ou plusieurs autres formats courants. Votre environnement de développement sait probablement gérer l'un de ces formats. Sur les ordinateurs personnels, Access, xBASE et MySQL sont les plus utilisés. Si les données que vous souhaitez utiliser sont dans l'un de ces formats, la conversion sera un jeu d'enfant. Sinon, il vous faudra procéder en deux étapes.

En dernier ressort, vous pourrez toujours vous retourner vers des professionnels de la conversion si le format de vos données est trop ancien, propriétaire, autant dire défunt. Ces gens se sont spécialisés dans les conversions et ils savent gérer des centaines de formats dont personne ou presque n'a jamais entendu parler. Donnez-leur une bande ou un disque qui contient les données à convertir, et ils vous les rendront dans le format de votre choix.

Transférer toutes les lignes entre des tables

Importer des données depuis une table de votre base pour les combiner aux données d'une autre table est un problème bien moins délicat. Cette importation ne pose pas de difficultés si la structure de la seconde table est identique à celle de la première, c'est-à-dire si chaque colonne de la première table dispose d'une colonne correspondante dans la seconde, et si les types de données de ces deux colonnes sont sinon identiques du moins compatibles. Si tel est le cas, vous pouvez combiner les données des deux tables en utilisant l'opérateur relationnel `UNION`. Le résultat produit sera une *table virtuelle* qui sera la somme des deux tables. Je traite des opérateurs relationnels, dont `UNION`, au Chapitre 11.

Transférer certaines colonnes et

certaines lignes entre des tables

La plupart du temps, les données de la table source n'adoptent pas exactement la structure de la table vers laquelle vous voulez les transférer. Il se peut que seules certaines colonnes se retrouvent dans la structure de la table destination, et ce sont précisément ces colonnes que vous voulez transférer. En combinant les opérateurs `SELECT` et `UNION`, il est possible de spécifier les colonnes de la table source qui seront incluses dans la table virtuelle résultante. En utilisant des clauses `WHERE` dans les instructions `SELECT`, vous pouvez conditionner le transfert des lignes à certaines conditions. Je traite dans le détail de la clause `WHERE` au Chapitre 10.

Supposons que vous disposiez de deux tables, PROSPECTS et CLIENTS, et que vous désiriez connaître la liste des personnes qui apparaissent dans les deux tables comme résidant dans le Maine. Vous pouvez créer une table virtuelle contenant les informations voulues en utilisant la commande suivante :

```
SELECT NOM, PRENOM
       FROM PROSPECTS
       WHERE ETAT = 'ME'
UNION
SELECT NOM, PRENOM
        FROM CLIENTS
        WHERE ETAT = 'ME' ;
```

Voyons cela de plus près :

- Les instructions `SELECT` spécifient que les colonnes incluses dans la table virtuelle résultat sont `NOM` et `PRENOM`.
- Les clauses `WHERE` limitent les lignes incluses à celles dont la colonne `ETAT` contient la valeur `'ME'`.
- La colonne `ETAT` n'est pas incluse dans la table de résultat, mais elle est présente dans les deux tables

sources.

» L'opérateur `UNION` combine les résultats produits par `SELECT` sur `PROSPECTS` avec les résultats produits par `SELECT` sur `CLIENTS`, supprime les lignes redondantes et affiche le résultat.

Une autre manière de copier les données d'une table vers une autre consiste à imbriquer une instruction `SELECT` dans une instruction `INSERT`. Cette méthode (une *sous-sélection* présentée au chapitre 12) ne crée pas de table virtuelle mais duplique les données sélectionnées. Vous pouvez prendre toutes les lignes de la table CLIENTS et les insérer dans la table PROSPECTS. Bien entendu, cela ne marche que si les structures des deux tables sont identiques. Si vous voulez par la suite placer dans la table PROSPECTS les seuls clients qui vivent dans le Maine, une simple instruction `SELECT` comportant une seule condition dans la clause `WHERE` suffira à répondre à la question :

```
INSERT INTO PROSPECT
 SELECT * FROM CLIENTS
   WHERE ETAT = 'ME' ;
```

Bien que cette opération crée des données redondantes (vous stockez désormais des données relatives aux clients dans la table PROSPECTS et dans la table `CLIENTS`), il peut s'avérer utile de l'effectuer, car elle permet d'améliorer les performances. Faites cependant attention à la redondance. Pour maintenir la cohérence des données, n'oubliez pas de reproduire dans la seconde table toutes les opérations de suppression, de modification ou d'ajout auxquelles vous procédez dans la première. Un autre problème potentiel est le risque de duplication des clés primaires. Si un prospect préexistant possède une clé primaire (`ProspectID`) identique à celle d'un client (`ClientID`) que vous tentez d'ajouter à la table PROSPECTS, l'opération d'insertion échouera.

Modifier des données existantes

Dans ce monde, la seule chose permanente, c'est le changement. Si vous craignez la routine, il vous suffit d'attendre un peu. Et puisque le changement est la règle, il en va de même pour vos bases de données. Un client change d'adresse, le stock diminue (ce qui est heureux pour vous), le classement du championnat de foot évolue chaque semaine... Autant d'événements qui nécessitent une actualisation de votre base de données.

L'instruction `UPDATE` de SQL permet de modifier les données d'une table. À l'aide d'une seule instruction `UPDATE`, vous pouvez actualiser une, quelques-unes ou toutes les lignes d'une table. Cette instruction s'utilise de la manière suivante :

```
UPDATE nom_table
SET colonne_1 = expression_1, colonne_2 =
expression_2,
..., colonne_n = expression_n
[WHERE predicats] ;
```

La clause `WHERE` est facultative. Elle sert à spécifier les lignes qu'il faut mettre à jour. Si vous ne l'utilisez pas, toutes les lignes de la table seront modifiées. La clause `SET` fournit les nouvelles valeurs à affecter aux colonnes que vous modifiez.

Prenez l'exemple de la table `CLIENTS` du Tableau 6.1.

TABLEAU 6.1 La table CLIENTS

Nom	Ville	Indicatif	Téléphone
Abe	Abelson Springfield	(714)	555-1111
Bill Bailey	Decatur	(714)	555-2222
Chuck Wood	Philo	(714)	555-3333
Don Stetson	Philo	(714)	555-4444
Dolph Stetson	Philo	(714)	555-5555

La liste des clients est parfois modifiée – les gens déménagent, changent de numéro de téléphone, et ainsi de suite. Supposons qu'Abe Abelson quitte Springfield pour habiter à Kankakee (c'est un joli nom, vous ne trouvez pas ?). Vous pouvez mettre à jour son enregistrement dans la table en utilisant l'instruction UPDATE suivante :

```
UPDATE CLIENTS
SET VILLE = 'Kankakee', TELEPHONE = '666-6666'
WHERE NOM = 'Abe Abelson' ;
```

Cette instruction crée les modifications reprises dans le tableau 6.2.

TABLEAU 6.2 La table CLIENTS après un UPDATE sur une ligne.

Nom	Ville	Indicatif	Téléphone
Abe Abelson	Kankakee	(714)	555-1111
Bill Bailey	Decatur	(714)	555-2222
Chuck Wood	Philo	(714)	555-3333
Don Stetson	Philo	(714)	555-4444
Dolph Stetson	Philo	(714)	555-5555

Vous pouvez utiliser une instruction similaire pour modifier plusieurs lignes. Supposons que la population de la localité Philo soit en train d'exploser. Du coup, la ville a désormais besoin de son propre code local. Vous pouvez appliquer une modification à tous les clients qui vivent à Philo à l'aide d'une seule instruction UPDATE :

```
UPDATE CLIENTS
     SET CODE LOCAL = '(619)'
     WHERE VILLE = 'Philo' ;
```

La table ressemble maintenant à celle reprise dans le Tableau 6.3.

Il est encore plus facile de mettre à jour en une seule fois toutes les lignes de la table. Nul besoin d'utiliser la clause `WHERE` pour limiter les effets de l'instruction. Supposons que la ville de

TABLEAU 6.3 La table CLIENTS après un UPDATE sur plusieurs lignes.

Nom	Ville	Indicatif	Téléphone
Abe Abelson	Kankakee	(714)	666-6666
Bill Bailey	Decatur	(714)	555-2222
Chuck Wood	Philo	(619)	555-3333
Don Stetson	Philo	(619)	555-4444
Dolph Stetson	Philo	(619)	555-5555

Rantoul acquière une importance politique majeure au point d'annexer Kankakee, Decatur et Philo, ainsi que toutes les villes citées dans la base de données. Vous pouvez mettre à jour toutes les lignes de la manière suivante :

```
UPDATE CIENTS
SET VILLE = 'Rantoul' ;
```

Le Tableau 6.4 vous montre le résultat.

TABLEAU 6.4 La table CLIENTS après UPDATE de toutes les lignes.

Nom	Ville	Indicatif	Téléphone
Abe Abelson	Rantoul	(714)	666-6666
Bill Bailey	Rantoul	(714)	555-2222
Chuck Wood	Rantoul	(619)	555-3333
Don Stetson	Rantoul	(619)	555-4444

La clause `WHERE` que vous utilisez pour identifier les lignes auxquelles l'instruction `UDPATE` s'applique peut contenir une sous-sélection. Une sous-sélection vous permet de mettre à jour des lignes dans une table en fonction du contenu d'une autre table.

Supposons par exemple que vous soyez un grossiste et que votre base de données contienne une base VENDEURS enregistrant les noms de tous les producteurs auxquels vous achetez vos produits. Vous pouvez aussi avoir une table PRODUITS pour les noms de tous les produits que vous revendez et les prix de chacun. La table VENDEURS possède les colonnes `VENDEUR_ID`, `VENDEUR_NOM`, `RUE`, `VILLE`, `ETAT` et `CODE_POSTAL`. La table PRODUITS contient `PRODUIT_ID`, `PRODUIT_NOM`, `VENDEUR_ID` et `PRIX_VENTE`.

L'un de vos vendeurs, Cumulonimbus Corporation, décide d'augmenter le prix de tous ses produits de 10 %. Pour maintenir votre marge, vous devez remonter les prix auxquels vous vendez les produits de Cumulonimbus de 10 %. Vous pouvez le faire en une seule instruction `UPDATE` :

```
UPDATE PRODUITS
    SET PRIX_VENTE = (PRIX_VENTE * 1.1)
     WHERE VENDEUR_ID IN
     (SELECT VENDEUR_ID FROM VENDEURS
    WHERE VENDEUR_NOM = 'Cumulonimbus
Corporation') ;
```

La sous-sélection trouve le `VENDEUR_ID` de Cumulonimbus. Vous pouvez alors utiliser le champ `VENDEUR_ID` de la table PRODUITS pour rechercher toutes les lignes que vous devez mettre à jour. Les prix de tous les articles de Cumulonimbus seront augmentés de 10 %. Les autres tarifs ne seront pas modifiés. Je traite des sous-sélections plus en détail au Chapitre 12.

Transférer des données

En plus d'INSERT et UPDATE, vous disposez d'une instruction supplémentaire pour ajouter des données à une table ou à une vue : MERGE. Elle permet de fusionner des données à partir d'une table ou d'une vue vers une autre table ou vue. L'objectif peut être d'ajouter de nouvelles lignes ou d'effectuer une mise à jour. MERGE est intéressante pour prendre des données qui existent déjà dans une base pour les copier vers une nouvelle destination.

Prenons comme exemple la base de données VetLab décrite au Chapitre 5. Supposons que, dans le personnel du laboratoire, se trouvent des commerciaux chargés des ventes, d'autres qui ont simplement à traiter celles-ci, ainsi que des employés des services généraux. Les affaires de votre filiale américaine ont été bonnes l'année dernière, et vous voudriez partager les fruits de la croissance avec votre personnel. Vous décidez d'attribuer une prime de 100 euros aux commerciaux qui ont réalisé au moins une vente, et de 50 euros à tous les autres. Vous commencez par créer une table PRIMES. Vous y insérez ensuite un enregistrement pour chaque employé qui apparaît au moins une fois dans la table COMMANDES, en leur affectant une prime par défaut de 100 euros.

Vous voulez maintenant utiliser l'instruction MERGE pour insérer de nouveaux enregistrements concernant les autres employés. Le montant de leur prime est donc de 50 euros. Voyons le code qui permet de créer la table PRIMES et de la remplir :

```
CREATE TABLE PRIMES (
   NOM_EMPLOYE CHARACTER (30) PRIMARY KEY,
   PRIME NUMERIC DEFAULT 100 ) ;

INSERT INTO PRIMES (NOM_EMPLOYE)
    (SELECT NOM_EMPLOYE FROM EMPLOYES, COMMANDES
      WHERE EMPLOYES.NOM_EMPLOYE =
COMMANDES.COMMERCIAL
      GROUP BY EMPLOYES.NOM_EMPLOYE) ;
```

Vous pouvez maintenant interroger la table PRIMES pour voir ce qu'elle contient :

```
SELECT * FROM PRIMES ;
```

```
NOM_EMPLOYE        PRIME
-------------- ----------
Brynna Jones       100
Chris Bancroft     100
Greg Bosser        100
Kyle Weeks         100
```

Il est ensuite facile de donner une prime de 50 euros au reste des employés :

```
MERGE INTO PRIMES
     USING EMPLOYES
     ON (PRIMES.NOM_EMPLOYE =
EMPLOYES.NOM_EMPLOYE)
     WHEN NOT MATCHED THEN INSERT
(PRIMES.NOM_EMPLOYE, PRIMES.PRIME)
 VALUES (EMPLOYES.NOM_EMPLOYE, 50) ;
```

Vous insérez ainsi dans la table PRIMES des enregistrements pour les personnes de la table EMPLOYES qui ne sont pas déjà mémorisées dans ladite table PRIMES. Ces agents se voient attribuer une prime de 50 euros. Une requête sur la table PRIMES pourrait alors donner ceci :

```
SELECT * FROM PRIMES ;

NOM_EMPLOYE        PRIME
-------------- ----------
Brynna Jones       100
Chris Bancroft     100
Greg Bosser        100
Kyle Weeks         100
Neth Doze           50
Matt Bak            50
Sam Saylor          50
Nic Foster          50
```

Les quatre premiers enregistrements, ceux qui ont été créés par l'instruction INSERT, sont classés dans l'ordre alphabétique des noms. Les quatre autres, ajoutés via l'instruction MERGE, apparaissent dans l'ordre où ils figurent dans la table EMPLOYES.

Supprimer des données obsolètes

Le temps passe, et votre table peut finir par contenir des données qui ne sont plus utiles. Vous voudrez alors les supprimer, car elles occupent de la place et prêtent à confusion. Vous pourriez les transférer dans une table archive que vous rendriez inaccessible. De la sorte, vous pourrez toujours les récupérer si le besoin s'en fait sentir. Mais que vous décidiez d'archiver ou non des données obsolètes, il faudra les supprimer de la base. SQL vous permet d'effacer des lignes de vos tables à l'aide de l'instruction DELETE.

Vous pouvez détruire toutes les lignes d'une table en utilisant une seule instruction DELETE, ou limiter la suppression à quelques lignes en ajoutant une clause WHERE. La syntaxe que vous emploierez est semblable à celle de l'instruction SELECT, à ceci près que vous ne spécifiez aucune colonne. Si vous supprimez une ligne, vous supprimez à l'évidence toutes les données que contiennent ses colonnes !

Par exemple, supposons que votre client David Taylor vienne juste de déménager à Tahiti, si bien qu'il ne va plus jamais vous acheter quoi que ce soit. Vous pouvez le retirer de votre table CLIENTS en utilisant l'instruction suivante :

```
DELETE FROM CLIENTS
   WHERE PRENOM = 'David' AND NOM = 'Taylor' ;
```

En supposant qu'il n'existe qu'un seul client David Taylor, cette instruction effectuera la suppression attendue. Si vous avez deux clients qui se nomment David Taylor, vous devrez ajouter une clause WHERE (telle que RUE ou TELEPHONE, ou CLIENT_ ID) pour être bien certain que vous supprimez le bon client. Sinon, tous les David Taylor de votre table seront effacés, y compris ceux qui ne sont pas partis à Tahiti...

Chapitre 7
Gérer l'historique des données

DANS CE CHAPITRE :

- **Définir des périodes.**
- **Suivre à la trace ce qui se passe à certains instants.**
- **Fournir un historique des modifications apportées aux données.**
- **Savoir ce qui s'est passé et quand l'évènement a été enregistré.**

Avant SQL:2011, le standard SQL de l'ISO/IEC ne disposait d'aucun mécanisme pour gérer les données qui étaient valides à un instant donné mais invalides à un autre. Or, toute application qui a besoin de suivre à la trace l'état des données a besoin de cette fonctionnalité. Cela signifiait donc que la charge de conserver une trace de ce qui était valide à un instant donné retombait sur le programmeur de l'application plutôt que sur la base de données. Autant dire que c'était la porte ouverte à des applications compliquées, hors budget, en retard et infestées de bogues.

Une nouvelle syntaxe a été ajoutée dans SQL:2011 pour permettre de gérer des données temporelles sans avoir à intervenir dans le code qui gère les données sans tenir compte de la temporalité. C'est un gros avantage pour quiconque souhaite qu'une base de données SQL intègre la gestion des temporalités.

Qu'est-ce que j'entends par *données temporelles* ? LE standard SQL:2011 de l'ISO/IEC n'utilise pas ce terme, mais il est fréquent dans la communauté des bases de données. Dans SQL:2011, une donnée temporelle est une donnée à laquelle une ou plusieurs périodes de temps sont associées, durant lesquelles la donnée est censée être valide. Autrement dit, cela signifie qu'avec les données temporelles, vous pouvez déterminer si une certaine donnée est valide ou non.

Dans ce chapitre, je vous présente le concept de période, en le définissant d'une manière bien particulière. Vous découvrirez ensuite les différentes temporalités et les effets qu'elles peuvent avoir sur la définition de clés primaires et de contraintes d'intégrité référentielle. Pour finir, je présente la manière dont des données très complexes peuvent être stockées et traitées dans des tables bitemporelles.

Comprendre les périodes dans SQL:2011

Quoique les versions du standard SQL antérieures à SQL:2011 comportaient les types de données `TIME`, `TIMESTAMP`, et `INTERVAL`, aucun type de données ne recouvrait la notion de *période* commençant à un certain instant et se terminant à un autre. Une nouvelle manière d'adresser ce besoin est de définir un type de données `PERIOD`. Toutefois, SQL:2011 ne procède pas ainsi. Introduire un nouveau type de données dans SQL à cette étape tardive de son développement, c'était prendre le risque de remettre en cause l'écosystème qui s'était développé autour. Il aurait fallu des interventions chirurgicales très lourdes dans tous les produits de bases de données du marché pour ajouter un nouveau type de données.

En conséquence, plutôt que d'ajouter un type de données `PERIOD`, SQL:2011 résout le problème en ajoutant des *définitions de périodes* sous la forme de métadonnées aux tables. Une définition de période est un élément nommé d'une table, qui identifie une paire de colonnes contenant le moment initial et le moment final. Les commandes `CREATE TABLE` et `ALTER TABLE` utilisés pour créer et modifier des tables ont été mises à jour pour supporter une nouvelle syntaxe permettant de créer ou détruire de telles périodes.

Une période est déterminée par deux colonnes : une colonne de début et une colonne de fin. Ces colonnes sont conventionnelles, tout comme les colonnes de n'importe quel type de données temporel. Chacune porte un nom unique. Comme mentionné plus tôt, une définition de période est un élément nommé d'une table. Elle se trouve dans le même espace de noms que les noms des colonnes, si bien qu'elle ne doit pas porter le nom d'une colonne.

SQL suit un modèle fermé-ouvert pour les périodes, ce qui signifie qu'une période comprend un début mais pas de fin. Pour toute ligne de la table, la fin d'une période doit être supérieure à son début. C'est une contrainte dont le SGBD assure le respect.

Deux dimensions du temps sont à prendre en compte lorsque vous manipulez des données temporelles :

- **Le temps-valide** est la période durant laquelle une ligne de la table reflète bien la réalité.
- **Le temps-transaction** est la période durant laquelle la ligne est enregistrée dans la base de données.

Le temps-valide et le temps-transaction d'une ligne de la table n'ont pas besoin d'être identiques. Par exemple, dans une base de données qui enregistre la durée de validité d'un contrat, l'information sur le contrat peut être insérée avant que le contrat ne prenne effet (ce qui se fait d'ailleurs généralement).

Dans SQL:2011, des tables distinctes peuvent être créées et maintenues pour les deux types de temporalités, à moins d'utiliser une table bitemporelle (il en sera question plus loin dans ce chapitre). Les informations du temps-transaction sont conservées dans des tables qui contiennent une période temps-système, signalée par le mot-clé `SYSTEM_TIME`. Quant aux informations du temps-valide, elles sont maintenues dans des tables qui contiennent une période de temps-application. Vous pouvez donner n'importe quel nom à une période de temps-application, pourvu que le nom ne soit pas déjà utilisé ailleurs. Vous avez le droit de définir au plus une période de temps-système et une période de temps-application.

Quoique la gestion des données temporelles dans SQL a été introduite pour la première fois avec SQL:2011, les gens n'ont pas attendu qu'elle soit intégrée aux produits de bases de données pour se servir de données temporelles. On utilisait classiquement deux tables, une pour le début et une autre pour la fin de la période. Le fait que SQL:2011 n'introduise pas de type de données `PERIOD`, mais plutôt des définitions de période sous la forme de métadonnées, signifie que les tables utilisées jusqu'alors peuvent facilement être rendues compatibles avec le nouveau mécanisme. La logique

qui permet de fournir des informations sur une période peut être retirée des applications, ce qui permet de les simplifier, d'en améliorer les performances et d'en renforcer la fiabilité.

Travailler avec des tables de périodes de temps-application

Prenons un exemple utilisant des tables de périodes de temps-application. Considérons une entreprise qui veut conserver la trace temporelle des départements par lesquels sont passés les employés durant leur contrat. L'entreprise peut y parvenir en créant des tables de périodes de temps-application pour les employés et les départements, de la manière suivante :

```
CREATE TABLE employe_atpt(
EmpID INTEGER,
EmpDebut DATE,
EmpFin DATE,

EmpDept VARCHAR(30),
PERIOD FOR EmpPeriode (EmpDebut, EmpFin)
);
```

La date et l'heure de début (`EmpDebut` dans cet exemple) figurent dans la période, mais celles de fin (`EmpFin` dans cet exemple) n'y figurent pas. C'est cela, la sémantique fermé-ouvert.

Je n'ai pas encore précisé de clé primaire, car c'est un peu plus complexe lorsque vous traitez des données classiques. Je vais en parler à la fin de ce chapitre.

Pour l'heure, insérez quelques données dans la table pour voir à quoi cela ressemble :

```
INSERT INTO employe_atpt
VALUES (12345, DATE '2011-01-01', DATE '9999-12-
31',
'Ventes');
```

La table qui en résulte est représentée dans le tableau 7.1.

TABLEAU 7.1 Une table à période de temps-application ne contenant qu'une ligne

EmpID	EmpDebut	EmpFin	EmpDpt
12345	2011-01-01	9999-12-31	Ventes

La date de fin 9999-12-31 indique que le contrat de travail de l'employé dans l'entreprise ne s'est pas encore terminé. Pour plus de simplicité, je n'ai pas fait figurer les heures, les minutes et les secondes, et encore moins les fractions de seconde des exemples.

Supposons maintenant que, le 15 mars 2012, l'employé 12345 est temporairement affecté au département Ingénierie jusqu'au 15 juillet 2012, date à laquelle il retournera au département Ventes. Vous pouvez le mentionner en utilisant la commande UPDATE suivante :

```
UPDATE employe_atpt
FOR PORTION OF EmpPeriode
          FROM DATE '2012-03-15'
          TO DATE '2012-07-15'
SET EmpDept = 'Ingénierie'
WHERE EmpID = 12345;
```

Après la mise à jour, la table contient trois lignes, comme dans le tableau 7.2 :

TABLEAU 7.2 La table de périodes de temps-application après mise à jour.

EmpID	EmpDebut	EmpFin	EmpDpt
12345	2011-01-01	2012-03-15	Ventes
12345	2012-03-15	2012-07-15	Igénierie
12345	2012-07-15	9999-12-31	Ventes

En supposant que l'employé 12345 travaille toujours au département Ventes, la table rend compte fidèlement de son appartenance depuis le jour de l'an 2011 jusqu'à l'instant présent.

Si vous pouvez insérer de nouvelles données dans la table et mettre à jour des données qui y figurent, vous devriez pouvoir en supprimer des données. Toutefois, supprimer des données d'une table de périodes de temps-application peut s'avérer un peu plus complexe que supprimer des lignes dans une table normale, c'est-à-dire non temporelle. Par exemple, supposons que l'employé 12345, plutôt que d'être transféré au département Ingénierie le 15 mars 2012, quitte l'entreprise et soit réembauché le 15 juillet suivant. À la base, la table des périodes de temps-application ne contiendra qu'une ligne, comme dans le tableau 7.3.

TABLEAU 7.3 La table de périodes de temps-application avant une mise à jour ou une suppression.

EmpID	EmpDebut	EmpFin	EmpDpt
12345	2011-01-01	9999-12-31	Ventes

Une commande `DELETE` va mettre à jour la table pour indiquer la période durant laquelle l'employé 12345 est parti :

```
DELETE employe_atpt
FOR PORTION OF EmpPeriode
          FROM DATE '2012-03-15'
          TO DATE '2012-07-15'
WHERE EmpID = 12345;
```

La table qui en résulte est reprise dans le tableau 7.4.

TABLEAU 7.4 La table de périodes de temps-application avant après une suppression.

EmpID	EmpDebut	EmpFin	EmpDpt
12345	2011-01-01	2012-03-15	Ventes

12345	2012-07-15	9999-12-31	Ventes

La table reflète mentionne maintenant les périodes durant lesquelles l'employé 12345 a été effectivement employé par l'entreprise, si bien qu'on note un vide durant la période où il l'avait quittée.

Vous aurez noté quelque chose d'intriguant avec les tables présentées dans cette section. Dans le listing d'une table non temporelle des employés d'une entreprise, le numéro d'identifiant d'un employé suffit pour servir de clé primaire, car il identifie de manière unique un employé. Toutefois, une table de périodes de temps-application peut contenir plusieurs lignes pour un même employé. En conséquence, l'identifiant d'un employé ne peut plus être utilisé comme clé primaire. Les données temporelles doivent lui être associées.

Concevoir des clés primaires dans une table de périodes de temps-application

Dans les tables 7.2 et 7.4, il est clair que l'identifiant de l'employé (`EmpID`) ne peut pas garantir l'unicité. En effet, on peut trouver plusieurs lignes avec le même `EmpID`. Pour garantir qu'aucune ligne n'est dupliquée, le début (`EmpDebut`) et la fin (`EmpFin`) de la période doivent figurer dans la clé primaire. Toutefois, il ne suffit pas de les y inclure. Jetez un œil sur le tableau 7.5, qui présente un cas où l'employé 12345 a été transféré au département Ingénierie durant quelques mois, avant de rejoindre son département d'origine.

TABLEAU 7.5 Une situation que vous souhaiterez éviter.

EmpID	EmpDebut	EmpFin	EmpDpt
1245	2011-01-01	9999-12-31	Ventes
1245	2012-03-15	2012-07-15	Ingénierie

Les deux lignes de la table sont garanties d'être uniques du fait que la clé primaire comprend EmpDebut et EmpFin, mais notez que les deux périodes se chevauchent. Il semble que l'employé 12345 fasse partie des deux départements Ventes et Ingénierie du 15 mars 2012 au 15 juillet 2012. Dans quelques entreprises, cela peut être possible, mais cela ajoute à la complexité et peut corrompre les données. Dans la plupart des entreprises, on veillera au respect d'une contrainte imposant qu'un employé ne puisse faire partie que d'un département à la fois. Vous pouvez ajouter une telle contrainte à la table avec une commande ALTER TABLE telle que la suivante :

```
ALTER TABLE employe_atpt
ADD PRIMARY KEY (EmpID, EmpPeriode WITHOUT
OVERLAPS);
```

Il existe une meilleure manière de parvenir au résultat que de créer la table avant d'y ajouter la contrainte de clé primaire. En effet, vous pouvez inclure la contrainte de clé primaire dans l'instruction `CREATE` initiale. Cela pourrait ressembler à :

```
CREATE TABLE employe_atpt
EmpID INTEGER NOT NULL,
EmpDebut DATE NOT NULL,
EmpFin DATE NOT NULL,
EmpDept VARCHAR(30),
          PERIOD FOR EmpPeriode (EmpDebut,
EmpFin)
          PRIMARY KEY (EmpID, EmpPeriode WITHOUT
OVER-
LAPS)
);
```

Dorénavant, les lignes dont les périodes se chevauchent sont interdites. Tant que j'y étais, j'ai ajouté des contraintes `NOT NULL` à tous les éléments de la clé primaire. Une valeur nulle affectée à l'un quelconque de ces champs sera une source d'erreurs maintenant. Normalement, le SGBD devrait s'en occuper, mais pourquoi prendre le risque ?

Appliquer des contraintes référentielles à une table de périodes de temps-application

Toute base de données censée contenir plus qu'une simple liste va probablement requérir plus d'une table. Si tel est le cas, les relations entre les tables doivent être définies, et des contraintes d'intégrité référentielles mises en place.

Dans l'exemple de ce chapitre, vous disposez d'une table de périodes de temps-application. Il existe une relation d'un à plusieurs entre la table des départements et la table des employés, car un département peut comprendre plusieurs employés, mais un employé ne peut appartenir qu'à un seul département. Cela signifie que vous devez rajouter une clé étrangère à la table des employés, qui référence la clé primaire de la table des départements. Ceci à l'esprit, créez de nouveau la table des employés, cette fois en utilisant une commande `CREATE` plus complexe. Créez la table des départements de la même manière.

```
CREATE TABLE employe_atpt (
EmpID INTEGER NOT NULL,
EmpDebut DATE NOT NULL,
EmpFin DATE NOT NULL,
EmpNom VARACHAR (30),
EmpDept VARCHAR (30),
PERIOD FOR EmpPeriode (EmpDebut, EmpFin)
PRIMARY KEY (EmpID, EmpPeriode WITHOUT OVERLAPS)
FOREIGN KEY (EmpDept, PERIOD EmpPeriode)
REFERENCES dept_atpt (DeptID, PERIOD DeptPeriode)
);

CREATE TABLE dept_atpt (
DeptID VARCHAR (30) NOT NULL,
Manager VARCHAR (40) NOT NULL,
DeptDebut DATE NOT NULL,
```

```
DeptFin DATE NOT NULL,
PERIOD FOR DeptDuree (DeptDebut, DeptFin),
PRIMARY KEY (DeptID, DeptDuree WITHOUT OVERLAPS)
);
```

Faire des requêtes sur une table de périodes de temps-application

Maintenant, des informations détaillées peuvent être extraites de la base de données en utilisant des commandes `SELECT` qui exploitent les périodes.

Par exemple, vous pourriez souhaiter récupérer la liste de tous les employés qui travaillent actuellement dans l'entreprise. Avant SQL:2011, vous deviez écrire une commande ressemblant à cela :

```
SELECT *
FROM employe_atpt
WHERE EmpDebut <= CURRENT_DATE()
          AND EmpFin > CURRENT_DATE();
```

Avec la nouvelle syntaxe, vous pouvez obtenir le même résultat plus facilement, comme suit :

```
SELECT *
FROM employe_atpt
WHERE EmpPeriode CONTAINS CURRENT_DATE();
```

Vous pouvez aussi récupérer la liste des employés qui ont travaillé dans l'entreprise sur une période donnée, comme suit :

```
SELECT *
 FROM employe_atpt
 WHERE EmpPeriode OVERLAPS
 PERIOD (DATE ('2012-01-01'), DATE ('2012-09-
16'));
```

Hormis `CONTAINS` et `OVERLAPS`, vous pouvez utiliser d'autres prédicats comme `EQUALS`, `PRECEDES`, `SUCCEEDS`, `IMMEDIATLY PRECEDES`, et `IMMEDIATLY SUCCEEDS`.

Ces prédicats fonctionnent ainsi :

- Si une période `EQUALS` une autre, alors elles sont strictement identiques.
- Si une période `PRECEEDS` une autre, alors elle se déroule avant.
- Si une période `SUCCEEDS` une autre, alors elle se déroule après.
- Si une période `IMMEDIATLY PRECEEDS` une autre, alors elle se déroule juste avant et y est contigüe.
- Si une période `IMMEDIATLY SUCCEEDS` une autre, alors elle se déroule juste après et y est contiguë.

Travailler avec des tables de versions systèmes

Les tables de versions systèmes servent à d'autres usages que les tables de périodes de temps-application, et fonctionnent en conséquence différemment. Les tables de périodes de temps-application vous permettent de définir des périodes et de traiter les données associées à un instant donné. Les tables de versions systèmes sont conçues pour créer un historique de l'ajout, de la modification ou de la suppression de données de la base de données. Par exemple, il est important pour une banque de savoir exactement quand un retrait ou un dépôt a été effectué, et cette information doit être conservée durant un certain temps prévu par la loi. De même, des traders ont besoin de conserver la trace de la date de leurs achats et de leurs ventes. On trouve bien d'autres cas à citer, où il est important de conserver la trace d'un événement, à la fraction de seconde près.

Les applications telles que les logiciels de banque ou de trading ont des exigences strictes :

- Toute opération de mise à jour ou de suppression doit sauvegarder l'état original de la ligne avant de le modifier.
- Le système, plutôt que les utilisateurs, maintient le début et la fin des périodes dans les lignes.

  Les lignes qui ont fait l'objet d'une mise à jour ou d'une suppression restent dans la table et sont dès lors qualifiées de *lignes historiques*. Les utilisateurs ne peuvent pas modifier le contenu de ces lignes ou les périodes qui leur sont associées. Seul le système le peut. Il le fait lors d'une mise à jour de colonnes qui ne correspondent pas à des périodes dans la table, ou lorsqu'une ligne est supprimée.

  Ces contraintes garantissent que l'historique des données ne peut être altéré, ce qui permet de répondre aux exigences des auditeurs et aux règles des autorités de régulation.

Les tables de versions systèmes se distinguent des tables de périodes de temps-application sur plusieurs points dans la commande `CREATE` qui permet de créer les unes et les autres :

- Alors dans une table de périodes de temps-application, l'utilisateur peut nommer la période comme il l'entend, elle doit être toujours nommée `SYSTEM_TIME` dans une table de versions systèmes.
- La commande `CREATE` doit comprendre les mots-clés `WITH SYSTEM VERSIONING`. Quoique SQL:2011 permette que le type de données du début et de la fin de la période soit du type `DATE` ou du type `TIMESTAMP`, vous préférez

toujours utiliser `TIMESTAMP`, car cela vous procure bien plus de précision que celle du jour. Bien entendu, le type adopté pour le début d'une période doit être identique à celui de la fin.

Pour illustrer l'utilisation des tables de versions systèmes, je vais prolonger l'exemple des employés et des départements. Vous pouvez créer une table de versions systèmes avec le code suivant :

```
CREATE TABLE employe_sys (
 EmpID INTEGER,
 Sys_Debut TIMESTAMP(12) GENERATED ALWAYS AS ROW
START,
 Sys_Fin TIMESTAMP(12) GENERATED ALWAYS AS ROW
END,
 EmpNom VARCHAR(30),
 PERIOD FOR SYSTEM_TIME (SysDebut, SysFin)
) WITH SYSTEM VERSIONING;
```

Une ligne dans une table de versions systèmes est considérée être une *ligne courante* du système si sa période de temps-système comprend l'instant présent. Autrement, elle est considérée comme une ligne historique.

Les tables de versions systèmes sont très semblables aux tables de périodes de temps-application sur de nombreux points, mais elles en différent sur d'autres. En particulier :

- Les utilisateurs ne peuvent affecter ou modifier les valeurs des colonnes Sys_Debut et Sys_Fin. Ces valeurs sont affectées et modifiées automatiquement par le SGBD. Cette situation est imposée par les mots-clés `GENERATED ALWAYS`.
- Lorsque vous utilisez la commande `INSERT` pour ajouter quelque chose dans une table de versions systèmes, la valeur de la colonne Sys_Debut prend automatiquement

pour valeur l'instant de la transaction. La valeur affectée à la colonne Sys_Fin est la plus grande valeur du type de données de cette colonne.

- Dans les tables de versions systèmes, les commandes `UPDATE` et `DELETE` agissent sur les lignes courantes. Les utilisateurs ne peuvent pas modifier ou supprimer des lignes historiques.
- Les utilisateurs ne peuvent pas modifier le début ou la fin de la période de temps-système des lignes, qu'elles soient courantes ou historiques.
- Lorsque vous utilisez une commande `UPDATE` ou `DELETE` sur une ligne courante, une ligne historique est automatiquement insérée.

  Une commande de mise à jour sur une table de versions systèmes insère d'abord une copie de l'ancienne ligne, la fin de sa période de temps-système devenant l'instant de la transaction. Cela indique que la ligne a cessé d'être une ligne courante. Ensuite le SGBD procède à la mise à jour, tout en passant le début de la période de temps-système à l'instant de la transaction. La ligne mise à jour est dorénavant une ligne courante. Les déclencheurs `UPDATE` des lignes vont être activés, mais ce ne sera pas le cas des déclencheurs `INSERT` quoique des lignes historiques aient été insérées.

Une commande `DELETE` sur une table de versions systèmes ne supprime pas physiquement les lignes concernées. Elle modifie leur fin de période de temps-système en y affectant l'instant courant du système. Cela indique que ces lignes ont cessé d'être courantes. Elles sont dorénavant des lignes

historiques. Lorsque vous effectuez une commande `DELETE`, les déclencheurs `DELETE` des lignes concernées sont activés.

Concevoir des clés primaires pour une table de versions systèmes

Il est bien plus facile de désigner les clés primaires dans des tables de versions systèmes que dans des tables de périodes de temps-application. En effet, vous n'avez pas à vous préoccuper des périodes. Dans les tables de versions systèmes, les lignes historiques ne peuvent pas être modifiées. Leur unicité était vérifiée lorsqu'elles étaient des lignes courantes. Mais comme elles ne peuvent plus être modifiées, il n'est plus nécessaire de s'en assurer.

Si vous ajoutez une contrainte de clé primaire à une table de versions systèmes à l'aide d'une commande `ALTER`, elle ne s'appliquera qu'aux lignes courantes, si bien que vous n'aurez donc pas à y faire figurer des informations relatives aux périodes dans la commande. Par exemple :

```
ALTER TABLE employe_sys
ADD PRIMARY KEY (EmpID);
```

Cela fait l'affaire. Simple et efficace.

Appliquer des contraintes d'intégrité référentielles à une table de versions systèmes

Il est tout aussi simple d'ajouter des contraintes d'intégrité référentielles à des tables de versions systèmes, pour les mêmes raisons. Voici un exemple :

```
ALTER TABLE employe_sys
ADD FOREIGN KEY (EmpDept)
          REFERENCES dept_sys (DeptID);
```

Seules les lignes courantes étant affectées, vous n'avez pas à mentionner les colonnes de début et de fin de période.

Faire des requêtes sur une table de versions systèmes

La plupart des requêtes sur des tables de versions systèmes cherchent à savoir quelles données sont valides à un instant donné ou sur une période donnée. Pour le permettre, SQL:2011 fournit de nouvelles syntaxes. Utilisez la syntaxe `SYSTEM_TIME AS OF` pour demander l'information valide à un instant donné. Par exemple, supposons que vous souhaitiez savoir qui a été employé par l'entreprise le 15 juillet 2013. Vous pourriez former la requête suivante :

```
SELECT EmpID, EmpNom, Sys_Debut, Sys_Fin
FROM employe_sys FOR SYSTEM_TIME AS OF
          TIMESTAMP '2013-07-15 00:00:00';
```

Cette commande retourne toutes les lignes dont le début de période est égal ou antérieur à l'instant spécifié, et dont la fin de période y est égale ou postérieure.

Pour savoir ce qui était valide sur une période donnée, vous pouvez utiliser une commande similaire, à l'aide de la syntaxe appropriée. Voici un exemple :

```
SELECT EmpID, EmpNom, Sys_Debut, Sys_Fin
FROM employe_sys FOR SYSTEM_TIME FROM
          TIMESTAMP '2013-07-01 00:00:00' TO
          TIMESTAMP '2013-08-01 00:00:00';
```

La requête vous retourne toutes les lignes débutant au premier instant indiqué, jusqu'au second, mais sans l'inclure. Vous pourriez autrement utiliser ceci :

```
SELECT EmpID, EmpNom, Sys_Debut, Sys_Fin
FROM employe_sys FOR SYSTEM_TIME BETWEEN
```

```
            TIMESTAMP '2013-07-01 00:00:00' AND
            TIMESTAMP '2013-07-31 24:59:59';
```

La requête vous retourne toutes les lignes débutant au premier instant indiqué, jusqu'au second, en l'incluant cette fois.

Si une requête sur une table de versions systèmes ne spécifie pas d'instant, son comportement pas défaut est de vous retourner toutes les lignes courantes du système. Le code pourrait ressembler à ceci :

```
SELECT EmpID, EmpNom, Sys_Debut, Sys_Fin
FROM employe_sys;
```

Si vous souhaitez récupérer toutes les lignes d'une table de versions systèmes, historiques et courantes, vous pouvez le faire en utilisant la syntaxe suivante :

```
SELECT EmpID, EmpNom, Sys_Debut, Sys_Fin
FROM employe_sys FOR SYSTEM_TIME FROM
          TIMESTAMP '2013-07-01 00:00:00' TO
          TIMESTAMP '9999-12-31 24:59:59';
```

Tracer encore plus les données avec des tables bitemporelles

Parfois, vous souhaitez savoir quand un événement s'est passé dans le monde réel et quand il a été enregistré dans la base de données. Dans ce cas, vous pourriez utiliser une table qui est à la fois une table de versions systèmes et une table de périodes de temps-application. Une telle table est qualifiée de bitemporelle.

Il existe toute une variété de cas où une table bitemporelle peut s'avérer utile. Par exemple, supposons que l'un de vos employés déménage et change alors d'état, passant de l'Oregon à Washington. Vous devez tenir compte du fait que les charges sur son salaire vont changer à la date officielle de son nouvel établissement. Toutefois, il n'est pas certain que la modification sera apportée dans la base de données ce jour même. Il faut donc enregistrer les deux instants, et une table bitemporelle fera cela parfaitement. La table de

périodes de temps-application conserve les périodes auxquelles le déménagement a été connu de la base de données, et la table de périodes de temps-application conserve les périodes auxquelles il a pris légalement effet. Voici un exemple de création d'une telle table :

```
CREATE TABLE employe_bt (
EmpID INTEGER,
EmpDebut DATE,
EmpFin DATE,
EmpDept Integer
PERIOD FOR EmpPeriode (EmpDebut, EmpFin),
Sys_Debut TIMESTAMP (12) GENERATED ALWAYS
          AS ROW START,
Sys_Fin TIMESTAMP (12) GENERATED ALWAYS
          AS ROW END,
EmpNom VARCHAR (30),
EmpRue VARCHAR (40),
EmpVille VARCHAR (30),
EmpEtatProvince VARCHAR (2),
EmpCodePostal VARCHAR (10),
PERIOD FOR SYSTEM_TIME (Sys_Debut, Sys_Fin),

PRIMARY KEY (EmpID, EPeriode WITHOUT OVERLAPS),
FOREIGN KEY (EDept, PERIOD EPeriode)
REFERENCES Dept (DeptID, PERIOD DPeriode)
) WITH SYSTEM VERSIONING;
```

Les tables bitemporelles servent à la fois de tables de versions systèmes et de tables de périodes de temps-application. L'utilisateur fournit des valeurs pour les colonnes de début et de fin de période. Une commande `INSERT` dans une telle table passe automatiquement la période de temps-système à la date et l'heure de la transaction. La valeur de la colonne de fin de la période de temps-système prend automatiquement la plus grande valeur possible pour son type de données.

Les commandes UPDATE et DELETE fonctionnent comme si la table était une table de périodes de temps-application. De même que dans le cas d'une table de versions systèmes, ces commandes n'affectent que les lignes courantes, une ligne historique étant automatiquement insérée à chaque fois.

Une requête sur une table bitemporelle peut spécifier une période de temps-système, une période de temps-application, voire les deux. Voici un exemple d'utilisation simultanée :

```
SELECT EmpID
FROM employe_bt FOR SYSTEM TIME AS OF
          TIMESTAMP '2013-07-15 00:00:00'
WHERE EmpID = 314159 AND
          EmpPeriode CONTAINS DATE '2013-06-20
00:00:00';
```

Chapitre 8
Spécifier des valeurs

DANS CE CHAPITRE :

- **Utiliser des variables pour éliminer du code redondant.**
- **Extraire une information fréquemment utilisée d'un champ d'une table.**
- **Combiner des valeurs élémentaires pour former des expressions complexes**

J'insiste tout au long de ce livre sur l'importance du rôle de la structure d'une base de données dans le maintien de l'intégrité de cette dernière. Cependant, vous ne devez jamais oublier que la chose la plus importante est la donnée elle-même. En effet, les valeurs que contiennent les cellules qui forment les intersections des lignes et des colonnes de la base de données sont la matière sur laquelle vous travaillez.

Vous pouvez représenter les valeurs de diverses manières. Vous pouvez les représenter directement ou les dériver de fonctions et d'expressions. Ce chapitre décrit les diverses formes de valeurs ainsi que les fonctions et les expressions.

Les *fonctions* examinent des données et calculent une valeur en fonction de ces dernières. Les *expressions* sont des combinaisons de données que SQL évalue pour produire une seule valeur.

Valeurs

SQL reconnaît différents types de valeurs :

- Les valeurs de lignes.
- Les valeurs littérales.

- Les variables.
- Les variables spéciales.
- Les références à des colonnes.

LES ATOMES NE SONT PAS INDIVISIBLES NON PLUS

Au XIX^{e} siècle, les scientifiques croyaient qu'un atome était le plus petit composant irréductible de la matière. C'est pourquoi ils l'appelèrent atome, qui vient du mot grec atomos qui veut dire indivisible. Aujourd'hui, les scientifiques savent que les atomes ne sont pas indivisibles. Ils sont formés de protons, de neutrons et d'électrons. À leur tour, les protons et les neutrons sont faits de quarks, de gluons et de quarks virtuels. Et peut-être bien que ces choses elles-mêmes ne sont pas indivisibles. Qui sait ?

La valeur d'un champ d'une table est qualifiée d'atomique, même si de nombreux champs ne sont pas indivisibles. Une valeur `DATE` est composée d'un mois, d'une année et d'un jour. Une valeur `TIMESTAMP` est composée d'une heure, de minutes, de secondes, et ainsi de suite. Une valeur `REAL` ou `FLOAT` est constituée d'un exposant et d'une mantisse. Une valeur `CHAR` est formée de composants auxquels vous pouvez accéder en utilisant `SUBSTRING`. Par conséquent, qualifier les champs d'une base de données d'atomiques est un abus de langage si l'on se reporte à la définition du terme, à savoir : le plus petit morceau de matière insécable.

Les valeurs de lignes

Les valeurs les plus visibles dans une base sont *les valeurs de lignes* des tables. Ce sont les données que chaque ligne d'une table contient. Une ligne est généralement composée de plusieurs éléments, car chaque colonne de la ligne contient une valeur. Un champ est l'intersection d'une seule colonne et d'une seule ligne. Un *champ* contient une valeur *scalaire* (ou *atomique*). Une valeur scalaire ou atomique n'est composée que d'elle-même.

Les valeurs littérales

En SQL, une *valeur* est représentée par une variable ou une constante. Contrairement à la valeur d'une *constante* (qui ne change jamais), la valeur d'une *variable* peut varier avec le temps. Comme quoi la logique du langage est parfois imparable.

Un type important de constante est la *valeur littérale*. Vous pouvez la considérer comme une valeur WYSIWYG, car What You See Is What You Get (ce que vous voyez correspond exactement à ce que vous obtenez). Autrement dit, la représentation est la valeur elle-même.

SQL dispose de nombreux types de données et donc de nombreux types de littéraux. Le Tableau 8.1 présente quelques exemples de littéraux de différents types.

TABLEAU 8.1 Des littéraux de différents types.

Type de données	Exemple de valeur littérale
BIGINT	8589934592
INTEGER	186282
SMALLINT	186
NUMERIC	186282.42
DECIMAL	186282.42
REAL	6.02257E-23
DOUBLE PRECISION	3.1415926535897E00

FLOAT	6.02257E-23
CHARACTER (15)	‘GRECE ‘
Note : Contient quinze caractères au total dont des espaces.	
VARCHAR (CHARACTER VARYING	‘grecque’
NATIONAL CHARACTER (15)	ELLAS ‘1
Note : Contient quinze caractères au total dont des espaces.	
NATIONAL CHARACTER VARYING	‘lepton’2
CHARACTER LARGE OBJECT (CLOB)	(une chaîne de caractères réellement très longue)
BINARY(4)	‘01001100011100001111000111001010’
VARBINARY(4) (BINARY VARYING(4))	‘01001100011100000’
BINARY LARGE OBJECT(512) (BLOB(512))	(une chaîne de uns et de zéros réellement très longue)
DATE	DATE ‘1969-07-20’
TIME (2)	TIME ‘13.41.32.50’
TIMESTAMP (0)	

	TIMESTAMP '1998-05-17-13.03.16.000000'
TIME WITH TIMEZONE (4)	TIME '13.41.32.5000-08.00'
TIMESTAMP WITH TIMEZONE (0)	TIMESTAMP '1998-05-17-13.03.16.0000+02.00'
INTERVAL DAY	INTERVAL '7' DAY

Remarquez que les littéraux de type non numérique sont entourés d'apostrophes. Ces délimiteurs permettent d'éviter toute confusion, mais posent parfois des problèmes.

Que se passe-t-il si une chaîne de caractères contient elle-même une apostrophe ? Vous devez alors doubler cette apostrophe pour montrer que l'un des deux signes fait partie de la chaîne et n'en marque donc pas la fin. Pour dire par exemple `'L'évaporation de l'eau'`, il vous faut écrire `' L''évaporation de l''eau '`.

Les variables

Les littéraux et autres types de constantes sont fort utiles, mais ils ne remplacent pas les variables. Dans bien des cas, se passer des variables revient à s'imposer une importante, et inutile, surcharge de travail. Une *variable* désigne tout simplement une quantité dont la valeur peut changer. L'exemple suivant devrait vous en démontrer l'utilité.

Supposons que vous soyez le distributeur d'un produit vendu à plusieurs classes de clients. Vous accordez le meilleur prix aux clients qui achètent des volumes importants, un prix un peu moins bon à ceux qui achètent des quantités moyennes, les personnes qui n'achètent qu'en faible quantité payant évidemment plein tarif. Vous voulez donc indexer vos prix sur les quantités achetées. Pour votre produit F-117A, vous décidez de facturer 1,4 fois le prix d'achat aux gros clients (classe C). Vous facturez 1,5 fois ce prix aux moyens acheteurs (classe B). Enfin, vous facturez 1,6 fois vos coûts aux petits clients (classe A).

Vous stockez les prix d'achat de vos produits et les tarifs auxquels vous les revendez dans une table que vous nommez TARIFS. Pour mettre en place

votre nouvelle structure de prix, vous allez exécuter les commandes SQL suivantes :

```
UPDATE TARIFS
  SET PRIX = COUT * 1.4
  WHERE PRODUIT = ‘F-117A’
   AND CLASSE = ‘C’ ;
UPDATE TARIFS
   SET PRIX = COUT * 1.5
   WHERE PRODUIT = ‘F-117A’
   AND CLASSE = ‘B’ ;
UPDATE TARIFS
   SET PRIX = COUT * 1.6
   WHERE PRODUIT = ‘F-117A’
   AND CLASSE = ‘A’ ;
```

Ce code répond exactement à vos besoins. Mais que se passe-t-il si un compétiteur agressif veut manger vos parts de marché ? Vous allez devoir réduire vos marges. Il vous faut alors saisir une commande telle que celle-ci :

```
UPDATE TARIFS
  SET PRIX = COUT * 1.25
  WHERE PRODUIT = ‘F-117A’
   AND CLASSE = ‘C’ ;
UPDATE TARIFS
  SET PRIX = COUT * 1.35
 WHERE PRODUIT = ‘F-117A’
   AND CLASSE = ‘B’ ;
UPDATE TARIFS
  SET PRIX = COUT * 1.45
  WHERE PRODUIT = ‘F-117A’
   AND CLASSE = ‘A’ ;
```

Si votre marché est très volatile, vous devrez souvent réécrire votre code SQL. Cela peut devenir rapidement fastidieux, en particulier si les prix

figurent à plusieurs endroits dans votre code. Pour résoudre ce problème, il suffit de remplacer les littéraux (tels que `1,45`) par des variables (telles que `coefficientA`). Vous pourriez alors procéder à vos mises à jour de la manière suivante :

```
UPDATE TARIFS
  SET PRIX = COST * :coefficientC
  WHERE PRODUIT = 'F-117A'
   AND CLASSE = 'C' ;
UPDATE TARIFS
  SET PRIX = COST * :coefficientB
  WHERE PRODUIT = 'F-117A'
   AND CLASSE = 'B' ;
UPDATE TARIFS
  SET PRIX = COST * :coefficientA
  WHERE PRODUIT = 'F-117A'
   AND CLASSE = 'A' ;
```

Dorénavant, chaque fois que les conditions du marché vous contraindront à modifier vos prix, vous n'aurez qu'à modifier les valeurs des variables `coefficientA`, `coefficientB` et `coefficientC`.

Vous verrez que les variables que vous utilisez de cette manière sont parfois nommées *paramètres* ou encore *variables hôtes*. Les variables sont appelées paramètres si elles figurent dans un module de code écrit en SQL, et variables hôtes si elles figurent dans du code SQL intégré.

Le *SQL intégré* correspond aux instructions SQL qui sont intégrées dans le code d'une application écrite dans un langage hôte. Une autre solution consiste à utiliser un module de langage SQL pour écrire tout un module de code SQL que l'application appellera. Le choix entre l'une et l'autre de ces méthodes dépend de votre implémentation.

Variables spéciales

Si l'utilisateur d'une machine client se connecte à une base de données hébergée par un serveur, cette connexion est nommée *session*. Si

l'utilisateur se connecte à plusieurs bases de données, la session associée à la connexion la plus récente est considérée comme étant la *session courante*. Les sessions *précédentes* sont dites *dormantes*. SQL définit plusieurs *variables spéciales* qui se révèlent fort utiles sur un système multiutilisateur. Elles conservent en effet la trace des différents utilisateurs. Voyons cela de plus près :

- `SESSION_USER` : La variable spéciale `SESSION_USER` contient une valeur égale à l'identifiant de l'utilisateur de la session SQL courante. Si vous écrivez un programme qui supervise la base de données, vous pouvez interroger `SESSION_USER` pour savoir qui est en train d'exécuter des instructions SQL.
- `CURRENT_USER` : Un module SQL peut disposer d'un identificateur d'autorisation pour un utilisateur donné. Cette valeur est stockée dans `CURRENT_USER`. Si le module n'a pas d'identifiant de ce type, la valeur de `CURRENT_USER` est celle de `SESSION_USER`.
- `SYSTEM_USER` : La variable `SYSTEM_USER` contient l'identifiant d'un utilisateur du système d'exploitation. Cet identifiant est éventuellement différent de celui qui Figure dans un module SQL. Un utilisateur peut se connecter au système sous le nom de `LARRY`, mais déclarer s'appeler `RESPONSABLE` dans un module. La valeur de `SESSION_USER` est alors `RESPONSABLE`. Si l'utilisateur ne fait aucune référence explicite à l'identificateur du module, et si `CURRENT_USER` contient aussi `RESPONSABLE`, alors la valeur de `SYSTEM_USER` est `LARRY`.

Les variables `SESSION_USER`, `SYSTEM_USER` et `CURRENT_USER` servent à suivre à la trace les accès au système. Vous pouvez tenir un journal de bord en stockant périodiquement les valeurs de ces variables

dans une table. Par exemple :

```
INSERT INTO LOGUTIL (SNAPSHOT)
   VALUES ('Utilisateur ' || SYSTEM_USER ||
          ' avec ID ' || SESSION_USER ||
' activé à ' || CURRENT_TIMESTAMP) ;
```

Cette instruction produit l'entrée suivante :

```
Utilisateur LARRY avec ID RESPONSABLE activé à
2006-05-
17-14.18.00
```

Références de colonnes

Les colonnes contiennent une valeur pour chaque ligne d'une table. Les instructions SQL se réfèrent souvent à ces valeurs. Une référence complète à une colonne se compose du nom de la table, d'un point et du nom de la colonne (comme dans `TARIFS. PRODUIT`). Regardez l'exemple suivant :

```
SELECT TARIFS.COUT
   FROM TARIFS
   WHERE TARIFS.PRODUIT = 'F-117A' ;
```

`TARIFS. PRODUIT` est une référence de colonne. Elle contient la valeur `'F117-A'`. `TARIFS. COUT` est aussi une référence de colonne, mais vous ne connaîtrez sa valeur que lorsque l'instruction `SELECT` aura été exécutée.

Seule la référence à des colonnes qui se trouvent dans la table courante a véritablement un sens. Vous pouvez donc généralement omettre de mentionner le nom de cette table. Ainsi, l'instruction suivante équivaut à la précédente :

```
SELECT COUT
  FROM TARIFS
```

```
  WHERE PRODUIT = 'F-117A' ;
```

Il peut cependant arriver que vous ayez à manipuler plusieurs tables. Il arrive que deux tables de la base de données contiennent des colonnes portant le même nom. Si tel est le cas, vous devez utiliser des noms de colonnes totalement qualifiés pour les discriminer.

Par exemple, supposons que votre société dispose de bureaux à Kingston et Jefferson et que vous gériez une liste des employés pour chaque site. Vous nommez `EMP_KINGSTON` la table des employés de Kingston et `EMP_JEFFERSON` celle des employés de Jefferson. Vous voulez récupérer la liste des personnels dont les noms apparaissent dans ces deux tables. L'instruction `SELECT` suivante vous donne le résultat voulu :

```
SELECT EMP_KINGSTON.NOM, EMP_ KINGSTON.PRENOM
  FROM EMP_ KINGSTON, EMP_JEFFERSON
  WHERE EMP_ KINGSTON.EMP_ID =
EMP_JEFFERSON.EMP_ID ;
```

Puisque l'identifiant d'un employé est unique quel que soit le site où il travaille, vous pouvez utiliser cette clé comme lien entre les deux tables. La requête ne retourne que les noms des employés qui émargent aussi bien à Kingston qu'à Jefferson.

Les expressions de valeur

Une *expression* est *simple* ou *complexe*. Elle peut contenir des valeurs littérales, des noms de colonnes, des paramètres, des variables hôtes, des sous-requêtes, des connecteurs logiques et des opérateurs arithmétiques. Mais quel que soit son degré de complexité, une expression doit pouvoir se réduire à une seule valeur.

C'est pourquoi les expressions SQL sont aussi nommées *expressions de valeur*. Il est possible de combiner plusieurs expressions de valeur pour n'en former qu'une seule, du moment qu'elles se réduisent à des valeurs dont les types sont compatibles.

SQL dispose de cinq expressions de valeur :

- L'expression de valeur chaîne.

- L'expression de valeur numérique.
- L'expression de valeur date/heure.
- L'expression de valeur intervalle.
- L'expression de valeur conditionnelle.

Les expressions de valeur chaîne

La plus simple des expressions de valeur chaîne est une *chaîne de caractères*. Mais cette expression peut aussi être la référence à une colonne, une fonction d'ensemble, une sous-requête scalaire, une expression `CASE`, une expression `CAST` ou une expression de valeur chaîne complexe. Je traite des expressions `CAST` et `CASE` au Chapitre 9 et des sous-requêtes au Chapitre 12.

Un seul opérateur est disponible pour les expressions de valeur de chaîne : la *concaténation*. Vous pouvez concaténer n'importe laquelle des expressions mentionnées au début de cette section pour créer une expression de chaîne plus complexe. Une paire de lignes verticales (`||`) représente cet opérateur de concaténation. Le tableau suivant propose quelques exemples d'expression de valeur de chaîne :

Cette expression	**Produit**
'Beurre ' ‖ 'salé'	'Beurre salé'
'Haricots' ‖ ' ' ‖ 'verts'	'Haricots verts'
PRENOM ‖ ' ' ‖ NOM	'André Dupont'
B'1100111' ‖ B'01010011'	B'110011101010011'
'' ‖ 'Machin'	'Machin'
'Machin' ‖ ''	'Machin'
'Ma' ‖ '' ‖ 'ch' ‖ '' ‖ 'in'	'Machin'

Comme vous pouvez le constater, si vous concaténez une chaîne à une autre de longueur nulle, le résultat est identique à la chaîne d'origine.

Les expressions de valeur numérique

Les *expressions de valeur numérique* vous permettent d'appliquer les opérateurs d'addition, de soustraction, de multiplication et de division à des types de données numériques. L'expression doit pouvoir se réduire à une valeur numérique. Les éléments qui la composent peuvent être de différents types, mais tous doivent être numériques. La nature du résultat dépend des types de données des éléments qui constituent l'expression. Le standard SQL ne spécifie pas strictement le type produit par une combinaison de valeurs numériques. Tout dépend de votre plate-forme. Consultez la documentation fournie avec celle-ci pour en apprendre plus sur cette question.

Voici quelques exemples d'expressions de valeur numérique :

- `27`
- `49 + 83`
- `5 * (12 – 3)`
- `PROTEINE + GRAISSE + HYDRATECARBONE`
- `LONGUEUR/5280`
- `COUT * : coefficientA`

Les expressions de valeur date/heure

Les *expressions de valeur date/heure* permettent d'effectuer des opérations sur des dates et des heures. Ces expressions peuvent être composées d'éléments de types `DATE`, `TIME`, `TIMESTAMP` ou `INTERVAL`. Le résultat d'une expression de valeur date/heure est toujours de type date/heure (`DATE`, `TIME` ou `TIMESTAMP`). Par exemple, l'expression suivante donne la date qu'il sera dans une semaine :

```
CURRENT_DATE + INTERVAL ‘7’ DAY
```

Les heures sont au format UTC (Universal Time Coordinated), c’est-à-dire qu’elles sont exprimées relativement au méridien de Greenwich. Mais vous pouvez aussi spécifier un décalage afin de vous ajuster à un fuseau horaire de votre choix. La syntaxe qui suit exprime l’heure locale de votre système :

```
TIME ‘22.55.00’ AT LOCAL
```

Jetez un coup d’œil sur cette autre syntaxe, plus longue :

```
TIME ‘22.55.00’ AT TIME ZONE INTERVAL ‘-08.00’
HOUR TO
MINUTE
```

Cette expression définit l’heure comme étant celle de Portland dans l’Oregon, qui est inférieure de 8 heures à celle du méridien de Greenwich.

Les expressions de valeur intervalle

Si vous faites la différence entre deux valeurs de type date/ heure, vous obtenez un *intervalle*. Notez bien que SQL ne vous permet pas d’ajouter deux valeurs de type date/heure (cela n’aurait pas de sens). Si vous ajoutez ou soustrayez deux intervalles, vous obtenez un nouvel *intervalle*. Il est aussi possible de multiplier ou diviser des intervalles entre eux.

Souvenez-vous que SQL dispose de deux types d’intervalles : *année-mois* et *jour-heure*. Pour lever toute ambiguïté, vous devez spécifier quel type vous utilisez dans une expression. Par exemple, l’expression suivante utilise un intervalle année-mois pour calculer l’âge théorique de votre retraite :

```
(ANNIVERSAIRE_60 – DATE_COURANTE) YEAR TO MONTH
```

L’exemple ci-dessous produit un intervalle de 40 jours :

```
INTERVAL ‘17’ DAY + INTERVAL ‘23’ DAY
```

L'exemple qui suit fournit une approximation du nombre de mois durant lesquels une mère de cinq enfants est restée enceinte :

```
INTERVAL '9' MONTH * 5
```

Les intervalles peuvent être négatifs aussi bien que positifs. Ils peuvent également consister en des expressions de valeur ou des combinaisons d'expressions de valeurs qui se réduisent à des intervalles.

Les expressions de valeur conditionnelle

Comme son nom l'indique, la valeur d'une *expression de valeur conditionnelle* dépend d'une condition. Les expressions de valeur conditionnelle `CASE`, `NULLIF` et `COALESCE` sont bien plus complexes que les autres types d'expressions. En fait, elles sont si complexes que je n'ai pas assez de place pour en parler ici. Je les traite en détail au Chapitre 9.

Les fonctions

Une *fonction* est une opération simple ou modérément complexe que les commandes usuelles de SQL ne savent pas effectuer, mais qui peut se révéler fort utile. SQL fournit des fonctions qui effectuent des tâches que le code de l'application écrite dans le langage hôte (celui dans lequel vos instructions SQL sont incorporées) devrait autrement effectuer. SQL dispose de deux catégories principales de fonctions : les *fonctions d'ensemble* et les *fonctions de valeurs*.

Utiliser les fonctions d'ensemble

Les *fonctions d'ensemble* s'appliquent à des ensembles de lignes plutôt qu'à une ligne unique. Elles renvoient telle ou telle caractéristique du jeu de lignes courant. Un tel ensemble peut contenir des lignes de la table, ou encore des parties de lignes isolées par une clause `WHERE` (je traite en détail de la clause `WHERE` au Chapitre 10). Les programmeurs appellent

quelquefois les fonctions d'ensemble *fonctions d'agrégat*, car ces fonctions extraient des informations de plusieurs lignes, les traitent d'une quelconque manière et retournent une seule information. Cette réponse est le fruit de *l'agrégation* des données que contiennent les lignes.

Pour illustrer l'utilisation des fonctions d'ensemble, observez le Tableau 8.2. Il s'agit d'un tableau nutritionnel, les valeurs étant fournies pour 100 grammes d'aliments.

Une table nommée ALIMENTS contient les informations présentées dans le Tableau 8.2. Les champs vides contiennent la valeur `NULL`. Les fonctions d'ensemble `COUNT`, `AVG`, `MAX`, `MIN` et `SUM` vont vous fournir des informations intéressantes sur cette table.

COUNT

La fonction `COUNT` vous indique combien de lignes comporte la table, ou combien de lignes de la table répondent à certains critères. L'utilisation la plus triviale de la fonction est la suivante :

TABLEAU 8.2 Valeurs nutritives pour 100 g d'aliments.

Aliment	Calories	Protéines (g)	Graisse (g)	Hydrate carboné (g)
Amandes grillées	627	18,6	57,7	19,6
Asperge 20	2,2	0,2	3,6	
Banane crue	85	1,1	0,2	22,2
Boeuf, hamburger maigre	219	27,4	11,3	
Poulet, plat allégé	166	31,6	3,4	
Opossum, rôti	221	30,2	10,2	
Porc, jambon	394	21,9	33,3	

Haricots	111	7,6	0,5	19,8
Soda	39			10,0
Pain, blanc	269	8,7	3,2	50,4
Pain, blé entier	243	10,5	3,0	47,7
Brocolis	26	3,1	0,3	4,5
Beurre	716	0,6	81,0	0,4
Bonbons	367		0,5	93,1
Huile d'arachide 421	5,7	10,4	81,0	

```
SELECT COUNT (*)
  FROM ALIMENTS ;
```

Cette instruction vous retourne comme valeur 15, car elle dénombre toutes les lignes de la table ALIMENTS. L'exemple suivant produit le même résultat :

```
SELECT COUNT (CALORIES)
  FROM ALIMENTS ;
```

Comme chaque ligne de la table comporte une entrée dans la colonne CALORIES, la fonction retourne à nouveau 15. Mais si la colonne avait contenu des valeurs nulles, la fonction ne les aurait pas prises en compte.

L'instruction suivante retourne une valeur de 11, car 4 des 15 lignes de la table contiennent une valeur non définie dans la colonne HYDRATECARBONE :

```
SELECT COUNT (HYDRATECARBONE)
FROM ALIMENTS ;
```

Un champ d'une table peut contenir une valeur nulle pour diverses raisons. Le plus souvent, cette valeur n'est pas encore connue. À moins qu'elle n'ait pas encore été saisie. Si la valeur vaut zéro, il arrive que l'opérateur chargé

de la saisie ne renseigne pas le champ. Ce n'est pas une bonne habitude, car le zéro est une valeur bien définie, que vous pouvez utiliser dans vos calculs. La valeur nulle n'est pas définie, si bien que SQL n'en fait rien. Les phrases « Le soda ne contient pas de graisse » et « Je ne sais pas combien de graisse contient le soda » ne sont pas du tout équivalentes !

Vous pouvez aussi utiliser la fonction `COUNT` en combinaison avec la clause `DISTINCT` pour déterminer combien de valeurs distinctes existent dans une colonne. Par exemple :

```
SELECT COUNT (DISTINCT GRAISSE)
  FROM ALIMENTS ;
```

La réponse que retourne cette instruction est 12. Vous pouvez voir que 100 g d'asperges contiennent autant de graisse que100 g de bananes (0,2 g), et que 100 g de haricots contiennent autant de graisse que 100 g de bonbons sucrés (0,5 g). La table ne contient donc que 12 valeurs distinctes de graisse.

AVG

La fonction calcule et retourne la moyenne des valeurs de la colonne spécifiée. Bien entendu, vous ne pouvez utiliser la fonction `AVG` que sur des colonnes qui contiennent des valeurs numériques, comme par exemple dans :

```
SELECT AVG (GRAISSE)
  FROM ALIMENTS ;
```

Le résultat est 15,37. Ce nombre est grand à cause de la présence du beurre dans la table. Vous vous demandez peut-être quelle valeur produirait l'instruction si vous ne teniez pas compte de cet aliment. Pour le savoir, utilisez une clause `WHERE` de la manière suivante :

```
SELECT AVG (GRAISSE)
  FROM ALIMENTS
  WHERE ALIMENT <> 'Beurre' ;
```

La valeur moyenne tombe à 10,32 g pour 100 g d'aliment.

MAX

La fonction `MAX` vous retourne la valeur maximale trouvée dans la colonne spécifiée. L'instruction suivante renvoie 81 (la teneur en graisse de 100 g de beurre) :

```
SELECT MAX (GRAISSE)
  FROM ALIMENTS ;
```

MIN

La fonction `MIN` vous retourne la valeur minimale qui se trouve dans une colonne. L'instruction suivante renvoie 0,4, car la fonction ne traite pas les valeurs nulles comme des zéros :

```
SELECT MIN (CARBOHYDRATE)
  FROM ALIMENTS ;
```

SUM

La fonction `SUM` vous retourne la somme de toutes les valeurs d'une colonne. L'instruction suivante renvoie 3 924, ce qui correspond au total des calories des 15 aliments :

```
SELECT SUM (CALORIES)
  FROM ALIMENTS ;
```

Les fonctions de valeurs

Beaucoup d'opérations peuvent être appliquées dans des contextes différents. Comme vous avez besoin d'utiliser fréquemment ces opérations, vous pouvez les incorporer dans du code SQL sous la forme de fonctions de valeurs. SQL ne dispose que de peu de fonctions de valeurs si on le

compare à des SGBD tels qu'Access ou FoxPro, mais ce sont probablement les seules dont vous vous servirez régulièrement. SQL utilise les trois types de fonctions de valeurs suivants :

- Les fonctions de valeur chaîne.
- Les fonctions de valeur numérique.
- Les fonctions de valeur date/heure.
- Les fonctions de valeur intervalle

Les fonctions de valeur chaîne

Les fonctions de valeur chaîne reçoivent une chaîne de caractères en entrée et produisent une autre chaîne de caractères en sortie. SQL dispose de six fonctions de ce type :

- `SUBSTRING`
- `SUBSTRING SIMILAR`
- `SUBSTRING_REGEX`
- `TRANSLSATE_REGEX`
- `OVERLAY`
- `UPPER`
- `LOWER`
- `TRIM`
- `TRANSLATE`
- `CONVERT`

SUBSTRING

Utilisez la fonction SUBSTRING pour extraire une partie d'une chaîne source. La chaîne extraite est du même type que la chaîne source. Par exemple, si la chaîne source est du type CHARACTER VARYING, il en ira de même pour la sous-chaîne.

La syntaxe de SUBSTRING est la suivante :

```
SUBSTRING (valeur_chaîne FROM début [FOR
longueur])
```

La clause qui Figure entre crochets ([]) est optionnelle. La chaîne extraite de valeur_chaîne commence au caractère repéré par début et se prolonge sur longueur caractères. Si la clause FOR n'est pas mentionnée, l'extraction se poursuit jusqu'à la fin de la chaîne source. Par exemple :

```
SUBSTRING ('Pain, blé entier' FROM 7 FOR 6)
```

La chaîne extraite est 'blé en'. Elle commence au septième caractère et prend six caractères. SUBSTRING ne semble pas a priori être une fonction très intéressante. Si j'ai une valeur littérale 'Pain, blé entier', je n'ai pas besoin d'une fonction pour savoir à quoi correspondent les six caractères à partir du septième. Cependant, SUBSTRING est une fonction très utile, car la valeur qu'elle manipule n'a pas besoin d'être un littéral. Il peut s'agir de n'importe quelle expression qui correspond à une chaîne. Ainsi, il est parfaitement possible d'utiliser une variable qui se nomme objetaliment et qui prend des valeurs différentes selon le contexte. L'expression suivante permettrait d'extraire une chaîne d'objetaliment, et ce quelle que soit la valeur chaîne courante représentée par cette variable :

```
SUBSTRING (:objetaliment FROM 7 FOR 6)
```

Toutes les fonctions de valeur se ressemblent, en ce sens qu'elles peuvent manipuler des expressions ou des valeurs littérales.

Vous devez faire attention à quelques petites choses quand vous utilisez la fonction SUBSTRING. Vérifiez que la sous-chaîne que vous décrivez est bien incluse dans la chaîne source. Si vous demandez une extraction à partir du septième caractère, mais que la chaîne source n'en comporte que quatre,

la fonction vous retournera une valeur nulle. Par conséquent, vous devez toujours avoir une idée relativement précise de la forme de vos données avant d'appliquer SUBSTRING. Ne spécifiez pas non plus de valeurs négatives, car la fin d'une chaîne ne peut précéder son début.

Si une colonne est de type VARCHAR, vous ne savez pas a priori quelle est sa longueur pour une ligne donnée. Mais l'absence de cette information ne pose pas de problème à la fonction SUBSTRING. Si la longueur que vous spécifiez est trop importante, SUBSTRING vous renverra tout ce qu'elle a pu trouver. Elle ne retournera pas une erreur.

Par exemple, supposons que vous utilisiez l'instruction :

```
SELECT * FROM ALIMENTS
   WHERE SUBSTRING (ALIMENT FROM 7 FOR 12) =
'blé' ;
```

Cette instruction retourne la ligne qui correspond dans la table ALIMENTS au pain au blé entier, alors même que la colonne ALIMENT contient une valeur dont la longueur est inférieure à 18 caractères ('pain, blé').

Si un *opérande* (une valeur à partir de laquelle un opérateur dérive une autre valeur) de SUBSTRING contient une valeur nulle, la fonction retournera une valeur nulle.

SUBSTRING SIMILAR

Cette fonction est triadique (cela signifie qu'elle opère sur trois paramètres). Ces trois paramètres sont une chaîne de caractère source, une chaîne de motif (string pattern) et un caractère d'échappement. Elle utilise ensuite les motifs correspondants (en fonction des expressions régulières basées sur POSIX) pour extraire et retourner une chaîne résultat dans la chaîne de caractère source.

Deux instances du caractère d'échappement, chacune suivie d'un guillemet, servent à décomposer la chaîne de motif en trois parties. En voici un exemple :

Supposez que la chaîne de caractère source S soit : 'Il y a quatre-vingt-sept ans, nos pères ont fondé sur ce continent une nouvelle nation'. Supposez

ensuite que la chaîne de motif R soit ‘quatre ‘/”’sept’/”’années’, où la barre oblique est le caractère d’échappement.

Alors

```
SUBSTRING S SIMILAR TO R ;
```

retourne un résultat qui correspond au milieu de la chaîne de motif ‘sept’ dans notre exemple.

SUBSTRING_REGEX

SUBSTRING_REGEX recherche une chaîne de motif d’expression régulière XQuery et retourne une occurrence de la sous-chaîne correspondante.

Conformément au standard international ISO/IEC JTC 1/SC 32, la syntaxe de l’expression régulière d’une sous-chaîne est la suivante :

```
SUBSTRING_REGEX <left paren>
    <XQuery pattern> [ FLAG <XQuery option flag>
]
    IN <regex subject string>
    [ FROM <start position> ]
    [ USING <char length units> ]
    [ OCCURRENCE <regex occurrence> ]
    { GROUP <regex capture group> ] <right paren>
```

<XQuery pattern> est l’expression de chaîne de caractère dont la valeur est une expression régulière XQuery.

<XQuery option flag> est une chaîne de caractère optionnelle correspondant à l’argument $flags de la fonction [XQuery F & O] fn : match.

<regex subject string> est la chaîne de caractère recherchée qui devrait correspondre au <Xquery pattern>.

`<start position>` est une valeur numérique exacte optionnelle à l'échelle 0 qui indique la position du caractère où doit commencer la recherche (sa valeur par défaut est `1`).

`<char length units>` est `CHARACTERS` ou `OCTETS`, indiquant l'unité dans laquelle `<start position>` est mesurée. (Sa valeur par défaut est `CHARACTERS`).

`<regex occurrence>` est une valeur numérique exacte optionnelle à l'échelle 0, qui indique quelle occurrence d'une correspondance est demandée (Sa valeur par défaut est `1`).

`<regex capture group>` est une valeur numérique exacte optionnelle à l'échelle 0 qui indique le groupe de capture d'une correspondance souhaitée. (Sa valeur par défaut est `0`, ce qui indique l'occurrence entière).

Voici quelques exemples d'utilisation de `SUBSTRING_REGEX` :

```
SUBSTRING_REGEX ('\p{L}*' IN 'Just do
it.')='Just'
SUBSTRING_REGEX ('\p{L}*' IN 'Just do it.' FROM
2)= 'ust'
SUBSTRING_REGEX ('\p{L}*' IN 'Just do it.'
OCCURRENCE 2)
= 'do'
SUBSTRING_REGEX ( '(do) (\p{L}*' IN 'Just do it.'
GROUP
2) = 'it'
```

TRANSLATE_REGEX

`TRANSLATE_REGEX` recherche une chaîne de motif d'expression régulière XQuery et retourne la chaîne avec l'une ou toutes les occurrences de l'expression régulière XQuery remplacée par une chaîne de remplacement XQuery.

Conformément au standard international ISO/IEC JTC 1/SC 32, la syntaxe d'une translitération regex est la suivante :

```
TRANSLATE_REGEX <left paren>
<XQuery pattern> [ FLAG <XQuery option flag> ]
IN <regex subject string>
[ WITH <regex replacement string> ]
[ FROM <start position> ]
[ USING <char length units> ]
[ OCCURRENCE <regex transliteration occurrence> ]
<right
paren>

<regex transliteration occurrence> : :=
<regex occurrence>
| ALL
```

Où :

- `<regex replacement string>` est une chaîne de caractère dont la valeur convient à l'utilisation de l'argument `$replacement` de la fonction [`XQuery F & O] fn : replace`. Sa valeur par défaut est la chaîne zéro de longueur.
- `<regex transliteration occurrence>` est soit le mot clé `ALL` soit une valeur numérique exacte à l'échelle 0, indiquant quelle occurrence de la correspondance recherchée (par défaut `ALL`).

Voici quelques exemples avec une chaîne sans remplacement :

```
TRANSLATE_REGEX ('i' IN 'Bill did sit.') = 'Bll
dd st.'
TRANSLATE_REGEX ('i' IN 'Bill did sit.'
```

```
OCCURRENCE ALL) =
‘Bll dd st.’
TRANSLATE_REGEX (‘i’ IN ‘Bill did sit.’ FROM 5) =
‘Bill
dd st.’
TRANSLATE_REGEX (‘i’ IN ‘Bill did sit.’
Occurrence 2) =
‘Bill dd sit.’
```

Voici quelques exemples avec une chaîne de remplacement :

```
TRANSLATE_REGEX (‘i’ IN ‘Bill did sit.’ WITH ‘a’)
= ‘Ball
dad sat. ‘
TRANSLATE_REGEX (‘i’ IN ‘Bill did sit.’ WITH ‘a’
OCCUR-
RENCE ALL)= ‘Ball dad
                  sat.’
TRANSLATE_REGEX (‘i’ IN ‘Bill did sit.’ WITH ‘a’
OCCUR-
RENCE 2) = ‘Bill dad
                  sit.’
TRANSLATE_REGEX (‘i’ IN ‘Bill did sit.’ WITH ‘a’
FROM 5)
= ‘Bill dad sat.’
```

OVERLAY

OVERLAY remplace une sous-chaîne donnée d’une chaîne (spécifiée par une position de départ numérique donnée et par une longueur donnée) par une chaîne de remplacement. Quand la longueur spécifiée de la sous-chaîne est zéro, on ne supprime rien dans la chaîne originale. Mais la chaîne de remplacement est insérée dans la chaîne originale en commençant à la position de départ spécifiée.

UPPER

La fonction `UPPER` convertit une chaîne de caractères en majuscules, comme l'illustrent les exemples du tableau suivant :

Cette instruction	Retourne
`UPPER ('e. e. cummings')`	`'E. E. CUMMINGS'`
`UPPER ('Isaac Newton, Ph.D')`	`'ISAAC NEWTON, PH.D.'`

La fonction `UPPER` n'affecte pas les chaînes qui sont déjà en majuscules.

LOWER

La fonction `LOWER` convertit une chaîne de caractères en minuscules, comme l'illustrent les exemples du tableau suivant :

Cette instruction	Retourne
`LOWER ('TAXES')`	`'taxes'`
`LOWER ('E. E. Cummings')`	`'e. e. cummings'`

La fonction `LOWER` n'affecte pas les chaînes qui sont déjà en minuscules.

TRIM

La fonction `TRIM` supprime les blancs (ou d'autres caractère) qui se trouvent au début ou à la fin d'une chaîne. Voici quelques exemples d'utilisation de `TRIM` :

Cette instruction	Retourne
`TRIM (LEADING ' ' FROM ' sens ')`	`'sens '`
`TRIM (TRAILING ' ' FROM ' sens ')`	`' sens'`
`TRIM (BOTH ' ' FROM ' sens ')`	`'sens'`
`TRIM (BOTH 's' from 'sens')`	`'en'`

Le caractère par défaut supprimé est l'espace. La syntaxe suivante est donc légale :

```
TRIM (BOTH FRsOM ‘ sens ‘)
```

Vous obtenez le même résultat qu'avec le troisième exemple de la table.

TRANSLATE et CONVERT

Les fonctions `TRANSLATE` et `CONVERT` prennent une chaîne source dans un jeu de caractères et la convertissent en une chaîne dans un autre jeu de caractères. Par exemple, vous pourriez convertir une chaîne de l'anglais en kanji ou de l'hébreu en français. Les fonctions de conversion qui spécifient ces transformations dépendent de l'implémentation. Reportez-vous à votre propre système pour plus de détails.

Si traduire d'un langage dans un autre était aussi simple que d'invoquer la fonction SQL `TRANSLATE`, ce serait formidable. Malheureusement, ce n'est pas aussi facile. `TRANSLATE` se contente de convertir le jeu de caractères d'une chaîne. Par exemple, cette fonction peut convertir `‘Ellas’` en `‘Ellas’`, mais elle ne peut pas traduire `‘Ellas’` en `‘Grèce’`.

Les fonctions de valeur numérique

Les fonctions de valeur numérique prennent des données de différents types en entrée, mais elles génèrent toujours une valeur numérique en sortie. SQL dispose de quinze de ces fonctions :

- Position (`POSITION`)
- Fonction d'occurrences regex (`OCCURRENCES_REGEX`)
- Position regex (`POSITION_REGEX`)
- Extraction (`EXTRACT`)
- Longueur (`CHAR_LENGTH, CHARACTER_LENGTH, OCTET_LENGTH`)

- Cardinalité (`CARDINALITY`)
- Valeur absolue (`ABS`)
- Modulo (`MOD`)
- Logarithme naturel (`LN`)
- Exponentielle (`EXP`)
- Puissance (`POWER`)
- Racine carrée (`SQRT`)
- Valeur approchée basse (`FLOOR`)
- Valeur approchée haute (`CEIL`, `CEILING`)
- Fonction d'intervalle (`WIDTH_BUCKET`)

POSITION

`POSITION` recherche la chaîne cible spécifiée dans une chaîne source et retourne la position du premier caractère de la cible. Pour une chaîne de caractères, la syntaxe est la suivante :

```
POSITION (cible IN source [USING char length
units])
```

Vous pouvez spécifier une unité de longueur autre que CHARACTER, mais c'est rare. Si vous utilisez des caractères Unicode selon le type le caractère peut être de 8, 16, ou 32 bits de long. Si le caractère est de 16 ou 32 bits de long, vous pouvez explicitement spécifier 8 bits avec USING OCTETS.

Pour une chaîne binaire, la syntaxe est la suivante

```
POSITION (cible IN source)
```

Si la valeur de la cible est égale à une sous-chaîne de longueur identique des octets contigus dans la chaîne source, alors le résultat est supérieur au

nombre d'octets qui précède le début du premier comme sous-chaîne.

Le tableau suivant présente quelques exemples :

Cette instruction	Retourne
`POSITION ('P' IN 'Pain, blé entier')`	1
`POSITION ('Pai' IN 'Pain, blé entier')`	1
`POSITION ('b' IN 'Pain, blé entier')`	7
`POSITION ('bli' IN 'Pain, blé entier')`	0
`POSITION ('' IN 'Pain, blé entier')`	1
`POSITION ('01001001' IN '001100010100100100100110')`	2

Si la fonction ne trouve pas la chaîne cible, elle retourne la valeur zéro. Si la chaîne cible est vide (comme dans le dernier exemple de caractère), `POSITION` retourne toujours un. Si n'importe lequel des deux opérandes de la fonction est une valeur nulle, la fonction retourne une valeur nulle.

OCCURRENCES_REGEX

`OCCURRENCES_REGEX` est une fonction numérique qui retourne le nombre de correspondances d'une expression régulière dans une chaîne. Sa syntaxe est la suivante :

```
OCCURRENCES_REGEX <left paren>
<XQuery pattern> [ FLAG <XQuery option flag> ]

IN <regex subject string>
[ FROM <start position> ]
[ USING <char length units> ] <right paren>
```

En voici quelques exemples :

```
OCCURRENCES_REGEX ( 'i' IN 'Bill did sit.' ) = 3
```

```
OCCURRENCES_REGEX ( ‘i’ IN ‘Bill did sit.’ FROM
5) = 2
OCCURRENCES_REGEX ( ‘I’ IN ‘Bill did sit.’ ) = 0
```

POSITION_REGEX

POSITION_REGEX est une fonction numérique qui retourne la position de départ d’une correspondance ou en plus la fin d’une correspondance d’une expression régulière dans une chaîne. En voici la syntaxe :

```
POSITION_REGEX <left paren> [ <regex position
start or
after> ]
<XQuery pattern> [ FLAG <XQuery option flag> ]
IN <regex subject string>
[ FROM <start position> ]
[ USING <char length units> ]
[ OCCURRENCE <regex occurrence> ]
[ GROUP <regex capture group> ] <right paren>

<regex position start or after> : := START |
AFTER
```

Et quelques exemples :

```
POSITION_REGEX ( ‘i’ IN ‘Bill did sit.’ ) = 2
POSITION_REGEX ( START ‘i’ IN ‘Bill did sit.’ ) =
2
POSITION_REGEX ( AFTER ‘i’ IN ‘Bill did sit.’ ) =
3
POSITION_REGEX ( ‘i’ IN ‘Bill did sit.’ FROM 5) =
7
POSITION_REGEX ( ‘i’ IN ‘Bill did sit.’
OCCURRENCE 2 ) =
7
```

```
POSITION_REGEX ( 'I' IN 'Bill did sit.' ) = 0
```

EXTRACT

La fonction `EXTRACT` extrait un champ d'une date/heure ou d'un intervalle. Par exemple, l'instruction suivante retourne 08 :

```
EXTRACT (MONTH FROM DATE '2000-08-20')
```

CHARACTER_LENGTH

La fonction `CHARACTER_LENGTH` donne le nombre de caractères dans une chaîne de caractères. Par exemple, l'instruction suivante retourne 13 :

```
CHARACTER_LENGTH ('Opossum, rôti')
```

Comme je l'ai fait remarquer plus haut au sujet de la fonction `SUBSTRING`, cette fonction n'est pas particulièrement utile si ses arguments sont des littéraux tels que 'Opossum, rôti'. Je peux tout aussi bien écrire 13 que `CHARACTER_LENGTH ('Opossum, rôti')`. En fait, cette fonction ne se révèle intéressante que si vous lui transmettez une variable en argument.

OCTET_LENGTH

Dans la terminologie informatique, un octet est une suite de huit bits. Presque tous les ordinateurs utilisent des octets pour représenter les caractères alphanumériques simples. Mais des langues plus complexes (comme le chinois) nécessitent 16 bits pour coder leurs signes. La fonction `OCTET_LENGTH` compte et retourne le nombre d'octets dans une chaîne. S'il s'agit d'une suite de bits, la fonction renvoie la quantité d'octets nécessaire pour stocker tous ces bits. Si c'est une chaîne écrite en français (un octet par caractère), la fonction retourne le nombre de caractères qu'elle contient. Si la chaîne est écrite en chinois, elle retourne deux fois le nombre de caractères que contient la chaîne. La chaîne suivante :

```
OCTET_LENGTH ('Haricots')                    renvoie
```

donne comme résultat 11, car chaque caractère peut être codé sur un seul octet.

Certains jeux de caractères utilisent un nombre d'octets différent selon les caractères. En particulier, quelques jeux de caractères, qui sont des mélanges de caractères kanji et latins, se servent de *caractères d'échappement* pour basculer entre les deux alphabets. Par exemple, une chaîne formée de kanji et de latin pourrait contenir 30 caractères et requérir 30 octets si tous les caractères sont latins, 62 octets s'ils sont tous kanji (60 octets plus les symboles d'échappement de début et de fin) et 150 octets si c'est une alternance de caractères latins et kanji (car chaque caractère kanji requiert deux octets, plus un octet pour les caractères d'échappement de début et de fin). La fonction `OCTET_LENGTH` renvoie le nombre d'octets dont vous avez besoin pour stocker la chaîne.

CARDINALITY

La *cardinalité* est le nombre d'éléments d'un ensemble. Elle concerne donc les collections (tableaux, ensembles, etc.) dans lesquelles chaque élément est une valeur possédant un certain type de donnée. En voici un exemple d'utilisation :

```
CARDINALITY (EquipeFoot)
```

Cette fonction devrait normalement renvoyer 11, à moins qu'une expulsion n'ait déjà eu lieu... Un *tableau* est une collection ordonnée d'éléments. Un *ensemble multiple* est une collection non ordonnée d'éléments. Dans le cas d'une équipe de football, dont la composition varie à chaque match et où de surcroît un même joueur peut occuper des postes différents, les ensembles multiples sont mieux adaptés que les tableaux.

ARRAY_MAX_CARDINALITY

La fonction `CARDINALITY` retourne le nombre d'éléments dans le tableau ou l'ensemble multiple spécifié, ce qui ne vous indique pas le maximum de cardinalité qui était attribué au tableau. Or cette information peut vous être utile dans de nombreux cas. C'est pourquoi SQL:2011 a ajouté une nouvelle fonction `AR-RAY_MAX_CARDINALITY`. Comme vous pouvez le

supposer, elle retourne la cardinalité maximale du tableau spécifié. Il n'y a pas encore de cardinalité maximale pour les ensembles multiples.

TRIM_ARRAY

La fonction `TRIM` supprime le premier ou dernier caractère d'une chaîne, alors que la fonction `TRIM_ARRAY` supprime les derniers éléments d'un tableau.

Pour supprimer les trois derniers éléments du tableau ALIMENT, utilisez la syntaxe suivante :

```
TRIM_ARRAY (ALIMENT, 3)
```

ABS

La fonction `ABS` retourne la valeur absolue d'une expression numérique. Exemple :

```
ABS (-273)
```

renvoie 273.

MOD

La fonction `MOD` retourne le reste de la division entière de deux expressions numériques. Exemple :

```
MOD (3, 2)
```

renvoie 1 (modulo de 3 divisé par 2).

LN

La fonction `LN` retourne le logarithme naturel d'une expression numérique. Exemple :

```
LN (9)
```

renvoie quelque chose comme 2,197224577. Le nombre de chiffres décimaux dépend de l'implémentation.

EXP

La fonction `EXP` retourne l'exponentielle d'une expression numérique (c'est-à-dire la base des logarithmes naturels, *e*, à la puissance spécifiée par l'expression). Exemple :

```
EXP (2)
```

renvoie quelque chose comme 7,389056. Le nombre de chiffres décimaux dépend de l'implémentation.

POWER

La fonction `POWER` retourne la valeur de la première expression numérique élevée à la puissance indiquée par la seconde.

Exemple :

```
POWER (2, 8)
```

renvoie 256, soit 2 élevé à la puissance 8 (c'est le plus grand entier enregistrable dans un octet).

SQRT

La fonction `SQRT` retourne la racine carrée de l'expression numérique. Exemple :

```
SQRT (4)
```

renvoie 2, la racine carrée de 4.

FLOOR

La fonction `FLOOR` retourne la plus grande valeur entière immédiatement inférieure à celle de l'expression numérique.

Exemple :

```
FLOOR (3.141592)
```

renvoie 3.0.

CEIL ou CEILING

La fonction `CEIL` (ou `CEILING`) retourne la plus petite valeur entière immédiatement supérieure à celle de l'expression numérique. Exemple :

```
CEIL (3.141592)
```

renvoie 4.0.

WIDTH_BUCKET

La fonction `WIDTH_BUCKET`, qui est utilisée dans *le traitement d'application en ligne* (OLAP), comporte quatre arguments. Elle renvoie un nombre compris entre 0 et la valeur de l'argument final plus 1. Elle affecte le premier argument à un *partage régulier de l'intervalle* défini par le deuxième et le troisième argument. Si le premier argument n'appartient pas à cet intervalle, la fonction retourne 0 (plus petit que la limite inférieure) ou le dernier argument plus 1 (plus grand que la limite supérieure). Exemple :

```
WIDTH_BUCKET (PI, 0, 9, 5)
```

`PI` est une constante prédéfinie de SQL dont la valeur est 3,141592. Cette instruction partage l'intervalle allant de zéro à neuf en cinq segments égaux (chacun représente donc une longueur de deux unités). La valeur de PI se trouve dans le second « godet » (entre deux et quatre). La fonction renvoie donc la valeur 2.

Fonctions de valeur date/heure

SQL dispose de trois fonctions qui retournent des informations sur la date courante, l'heure courante ou les deux. `CUR-RENT_DATE` renvoie la date système, `CURRENT_TIME` donne l'heure courante, et

CURRENT_TIMESTAMP retourne la date et l'heure courantes. CURRENT_DATE n'accepte aucun argument. Par contre, CURRENT_TIME et CURRENT_TIMESTAMP prennent un argument. Celui-ci spécifie la précision des secondes. Les types de données date/heure sont décrits au Chapitre 2. Le concept de précision y est aussi précisé.

Le tableau suivant présente quelques exemples d'utilisation de ces fonctions :

Cette instruction	Retourne
CURRENT_DATE	2006-12-31
CURRENT_TIME (1)	08 : 36 : 57.3
CURRENT_TIMESTAMP (2)	2006 12 31 08 : 36 : 57.38

La date que CURRENT_DATE retourne est de type DATE et non CHARACTER. L'heure fournie par CURRENT_TIME est de type TIME, tandis que la combinaison date/heure donnée par CURRENT_TIMESTAMP est de type TIMESTAMP. Comme SQL récupère ces informations en interrogeant l'horloge de votre ordinateur, elles sont toujours correctes pour le fuseau horaire dans lequel celui-ci se trouve.

Si vous voulez traiter les dates et les heures sous la forme de chaînes de caractères afin d'utiliser les fonctions qui manipulent ces dernières, vous pouvez procéder à une conversion de type en utilisant l'expression CAST décrite au Chapitre 9.

Fonctions de valeur intervalle

Une fonction de valeur intervalle nommée ABS a été introduite dans SQL:1999. Elle est similaire à la fonction de valeur numérique ABS, mais elle opère sur des données de type intervalle et non de type numérique. ABS prend un seul opérande et retourne un intervalle de précision identique qui ne comporte pas de valeur négative. En voici un exemple :

```
ABS ( TIME '11:31:00' – TIME '12:31:00' )
```

Le résultat est

```
INTERVAL +’1:00:00’ HOUR TO SECOND
```

Chapitre 9
Expressions SQL avancées

DANS CE CHAPITRE :

- **Utiliser des expressions conditionnelles CASE.**
- **Convertir des données d'un type dans un autre.**
- **Gagner du temps de saisie en utilisant des expressions.**

Au Chapitre 2, SQL est décrit comme un *sous-langage de données*. En fait, sa seule fonction consiste à manipuler des données dans une base. SQL manque des nombreuses fonctionnalités d'un langage procédural conventionnel. Il en résulte que les développeurs doivent passer sans cesse de SQL à un autre langage pour contrôler le flux de l'exécution. Ces incessants va-et-vient ralentissent le développement et affectent les performances lors de l'exécution. Les problèmes de performances liés aux limitations de SQL ont conduit ses concepteurs à lui ajouter des fonctionnalités supplémentaires lors de chaque sortie d'une nouvelle version du standard. L'une des fonctionnalités récentes de SQL, l'expression CASE, permet de construire des structures conditionnelles.

Une autre fonctionnalité, l'expression CAST, facilite la conversion des données d'une table d'un type vers un autre (c'est ce que l'on appelle le *transtypage*). Une troisième fonctionnalité, l'évaluation simultanée de plusieurs valeurs, permet de traiter une liste de valeurs là où vous ne pouviez auparavant n'en traiter qu'une seule. Par exemple, si votre liste de valeurs est une série de colonnes d'une table, vous pouvez maintenant leur appliquer une opération en une fois en utilisant une syntaxe fort simple.

Les expressions conditionnelles CASE

Chaque langage informatique digne de ce nom comporte une expression conditionnelle. En fait, la plupart des langages en ont plusieurs. La plus courante des expressions conditionnelles est sans doute la structure `IF... THEN... ELSE... ENDIF.` Si la condition qui suit le mot clé `IF` est vérifiée, le bloc de commandes qui suit le mot clé `THEN` est exécuté. Si cette condition n'est pas vérifiée, c'est le bloc de commandes qui suit ELSE qui est exécuté. Le mot clé `ENDIF` signale la fin de la structure. Cette structure permet de gérer des décisions à une ou deux issues. Elle n'est pas adaptée aux décisions plus complexes.

La plupart des langages complets ont une instruction du genre `CASE` qui gère l'exécution de plusieurs tâches à partir d'autant de conditions.

SQL diffère des autres langages en ce sens que `CASE` est une expression et non une instruction. En tant que telle, `CASE` est simplement un élément d'une instruction et non une instruction en soi. Avec SQL, vous pouvez placer une expression `CASE` quasiment partout où une valeur peut être utilisée. Lors de l'exécution, l'expression `CASE` est remplacée par une valeur. L'instruction `CASE` que vous trouvez dans les autres langages n'est pas évaluée à une valeur, mais elle exécute un bloc d'instructions.

Vous pouvez utiliser l'expression `CASE` comme suit :

- **Utiliser l'expression avec des critères de recherche.** `CASE` recherche les lignes d'une table pour lesquelles les critères spécifiés sont vérifiés. Si `CASE` trouve une ligne pour laquelle ces critères sont évalués comme étant vrais, l'instruction qui contient l'expression `CASE` est appliquée à cette ligne.
- **Utiliser l'expression** `CASE` **pour comparer un champ d'une table à la valeur spécifiée**. Le résultat de l'instruction qui contient l'expression `CASE` dépend de la valeur que contient le champ.

Les sections qui suivent, « Utiliser CASE avec des critères de recherche » et « Utiliser CASE avec des valeurs », vous permettront de mieux appréhender ces concepts. Vous trouverez tout d'abord deux exemples d'utilisation de CASE avec des critères de recherche. Un de ces exemples recherche des valeurs dans une table et les modifie en fonction d'une certaine condition. La seconde section propose deux exemples d'utilisation de la forme de CASE avec des valeurs.

Utiliser CASE avec des critères de recherche

Une utilisation intéressante de l'expression CASE est la recherche dans une table de lignes qui vérifient un critère de recherche. Si vous utilisez CASE de cette manière, l'expression adoptera la syntaxe suivante :

```
CASE
   WHEN critere1 THEN resultat1

   WHEN critere2 THEN resultat2
...
   WHEN criteren THEN resultatn
   ELSE resultatx
END
```

CASE examine la première des *lignes qualifiées* (la première ligne qui répond au critère de la clause WHERE s'il y en a une) pour évaluer le `critere1`. Si cette condition est vérifiée, l'expression CASE prend la valeur `result1`. Sinon, l'expression CASE évalue le `critere2`. S'il est vérifié, l'expression CASE prend la valeur `result2`, et ainsi de suite. Si aucun des critères n'est vérifié, l'expression CASE prend la valeur `resultatx`. La clause ELSE est optionnelle. Si l'expression n'a pas de clause ELSE et qu'aucun des critères n'est vérifié, elle prend la valeur nulle. Une fois l'instruction SQL qui contient l'expression CASE appliquée

à la première ligne qualifiée de la table, elle traite la ligne suivante, et ce jusqu'à ce que la table soit entièrement parcourue.

Modifier des valeurs en fonction de critère

Vous pouvez utiliser une expression `CASE` dans une instruction SQL à la place de n'importe quelle valeur. Par exemple, `CASE` peut être incorporée dans une instruction `UPDATE` pour effectuer plusieurs modifications dans une table en fonction d'un certain critère. Considérez l'exemple suivant :

```
UPDATE ALIMENTS
  SET EVALGRAISSE = CASE
         WHEN GRAISSE < 1
              THEN 'très allégé'
         WHEN GRAISSE < 5
            THEN 'allégé'
         WHEN GRAISSE < 20
            THEN 'modérément allégé'
         WHEN GRAISSE < 50
            THEN 'très gras'
         ELSE 'attaque cardiaque'
       END;
```

Cette instruction évalue les conditions WHEN dans l'ordre jusqu'à ce que l'une soit vérifiée, après quoi elle ignore les autres conditions.

Le Tableau 8.2 du Chapitre 8 représente une table nutritionnelle pour 100 grammes de divers aliments. Une base de données qui contiendrait ces informations peut recevoir une colonne `EVALGRAISSE` qui donnerait au consommateur un aperçu rapide du taux de matière graisseuse de chaque aliment. Si vous appliquez l'instruction `UPDATE` à la table ALIMENTS du Chapitre 8, elle attribuera aux asperges une valeur très allégée et au poulet une valeur allégée. Les autres aliments seront classés dans la catégorie à haut risque 'attaque cardiaque'.

Eviter des conditions qui génèrent des erreurs

L'expression CASE permet aussi *d'éviter des exceptions* en contrôlant des conditions susceptibles de provoquer des erreurs.

Prenons pour exemple le salaire de commerciaux. Les entreprises qui paient leurs commerciaux à la commission rémunèrent souvent leurs nouveaux employés en leur versant un salaire qui anticipe sur l'évolution de leurs résultats. Dans l'exemple suivant, les nouveaux commerciaux reçoivent un fixe qui diminue lorsque la commission augmente.

```
UPDATE COMP_VENTES
   SET COMP = COMMISSION + CASE
     WHEN COMMISSION <> 0
      THEN FIXE / COMMISSION
     WHEN COMMISSION = 0
      THEN FIXE
END;
```

Si la commission d'un commercial vaut zéro (il débute), la structure utilisée dans l'exemple permet d'éviter une opération de division par zéro, ce qui engendrerait une erreur. Si la commission du commercial n'est pas nulle, son salaire total est égal à la commission plus le fixe, celui-ci étant calculé proportionnellement à la commission.

Toutes les expressions THEN d'une structure CASE doivent être du même type : toutes numériques, toutes des chaînes de caractères ou toutes des dates. Le résultat de l'expression CASE possède alors ce même type.

Utiliser CASE avec des valeurs

Vous pouvez utiliser une forme plus compacte de l'expression CASE si vous voulez comparer une valeur test à une série d'autres valeurs. Cette variante se révèle utile au sein d'une instruction SELECT ou UPDATE dans le cas d'une table contenant un nombre limité de valeurs possibles dans une colonne, et que vous voulez associer un certain résultat à chacune

de ces valeurs. Si vous utilisez CASE de cette manière, l'expression adoptera la syntaxe suivante :

```
CASE valeurt
WHEN valeur1 THEN resultat1

    WHEN valeur2 THEN resultat2
    ...
    WHEN valeurn THEN resultatn
    ELSE resultatx
END
```

Si la valeur test (valeurt) est égale à valeur1, l'expression prend la valeur resultat1. Si valeurt n'est pas égale à valeur1, mais égale à valeur2, l'expression prend la valeur resultat2. Le test se poursuit jusqu'à ce qu'une correspondance soit trouvée. Si aucune des valeurs de comparaison n'est égale à la valeur test, l'expression prend la valeur resultatx. Si la clause optionnelle ELSE n'est pas présente, et si aucune des comparaisons ne réussit, l'expression prend la valeur nulle.

Pour comprendre comment fonctionne cette forme de CASE, prenons l'exemple d'une table qui contient les noms et les grades de divers officiers. Vous voulez dresser la liste des noms précédés de l'abréviation correcte de leur grade. L'instruction suivante effectue ce travail :

```
SELECT CASE GRADE
        WHEN 'général' THEN 'Gén. '
        WHEN 'colonel' THEN 'Col. '
        WHEN 'lieutenant-colonel' THEN 'Lt. Col.
'
        WHEN 'major' THEN 'Maj. '
        WHEN 'capitaine' THEN 'Capt. '
        WHEN 'premier lieutenant' THEN '1er Lt. '
        WHEN 'second lieutenant' THEN 2d Lt. '
        ELSE NULL
    END;
```

```
        NOM
    FROM OFFICIERS ;
```

Le résultat pourrait se présenter ainsi :

```
Capt. Lestocade
Col. Durand
Gén. Napoléon
Maj. Désastre
        Delabase
```

Comme le grade du soldat Delabase n'est pas mentionné dans l'expression `CASE`, la clause `ELSE` laisse vide ce renseignement. Prenons un autre exemple. Supposez que le capitaine Lestocade soit promu au rang de major. Vous voulez donc mettre à jour la base de données OFFICIERS pour en tenir compte. La variable `nom_officier` a pour valeur 'Lestocade' et la variable `nouveau_ grade` contient un entier (4) qui correspond au nouveau grade de Lestocade, conformément au tableau suivant :

nouveau_grade	**Grade**
1	général
2	colonel
3	lieutenant-colonel
4	major
5	capitaine
6	premier lieutenant
7	second lieutenant
8	NULL

Vous pouvez enregistrer la promotion en utilisant le code SQL suivant :

```
UPDATE OFFICIERS
    SET GRADE = CASE :nouveau_grade
                    WHEN 1 THEN 'général'
```

```
                    WHEN 2 THEN 'colonel'
                    WHEN 3 THEN 'lieutenant-
colonel'
                    WHEN 4 THEN 'major'
                    WHEN 5 THEN 'capitaine'
                    WHEN 6 THEN 'premier
lieutenant'
                    WHEN 7 THEN 'second lieutenant'
                    WHEN 8 THEN NULL
          END
    WHERE LAST_NAME = :nom_officier ;
```

L'utilisation de `CASE` avec des valeurs peut aussi prendre une syntaxe alternative :

```
CASE
    WHEN valeurt = valeur1 THEN resultat1
    WHEN valeurt = valeur2 THEN resultat2
    ...
    WHEN valeurt = valeurn THEN resultatn
    ELSE resultatx
END
```

Un cas particulier : NULLIF

Les champs d'une base de données contiennent des valeurs bien définies, du moins si elles sont déterminées. En général, si la valeur d'un champ est inconnue, il possède la valeur nulle (NULL). Sous SQL, vous pouvez utiliser une expression `CASE` pour changer le contenu d'un champ en une valeur nulle. La nullité indique que vous ne connaissez plus la valeur de ce champ.

Supposez par exemple que vous êtes l'heureux propriétaire d'une compagnie aérienne basée aux États-Unis qui propose des vols entre la Californie du Sud et l'État de Washington. Jusqu'à tout récemment, les vols s'arrêtaient à l'aéroport international de San José pour faire le plein.

Malheureusement, on vous a retiré l'autorisation d'atterrir à San José. Vous devez désormais faire escale pour remplir vos avions de kérosène soit à l'aéroport de San Francisco, soit à celui d'Oakland. Pour l'instant, vous ne savez pas quels sont les vols qui vont s'arrêter dans ces aéroports. Mais vous savez qu'aucun d'entre eux ne s'arrêtera plus à San José. Votre base de données VOLS contient des informations importantes sur chacune de vos routes aériennes. Vous allez donc mettre à jour cette base en y supprimant toute référence à San José. L'exemple suivant propose une manière de procéder :

```
UPDATE VOLS
    SET ESCALE_KEROSENE =CASE
                              WHEN ESCALE_KEROSENE
= 'San
                         José'
                                   THEN NULL
                              ELSE ESCALE_KEROSENE
                              END  ;
```

Comme ce genre de situation se reproduit couramment, SQL propose une notation abrégée pour effectuer ce type de travail. L'exemple précédent pourrait se présenter ainsi en utilisant cette notation abrégée :

```
UPDATE VOLS
    SET ESCALE_KEROSENE = NULLIF(ESCALE_KEROSENE,
'San
José');
```

Vous pouvez lire cette expression comme « mettre à jour la base de données `VOLS` en passant la colonne `ESCALE_KEROSENE` à null, si la valeur de `ESCALE_KEROSENE` est San José ; autrement, ne rien faire ».

`NULLIF` se révèle encore plus pratique si vous convertissez des données que vous aviez accumulées pour les utiliser avec un langage de programmation standard tel que COBOL ou Fortran. Les langages de programmation standard ne connaissent pas la valeur null. C'est pourquoi ils traduisent le concept d'« inconnu » à l'aide de valeurs spéciales. Par exemple, une valeur `-1` peut représenter la valeur inconnue pour

SALAIRE et la chaîne de caractères « *** » peut figurer une valeur inconnue ou indéterminable de CODE_TRAVAIL. Si vous voulez transcrire ces états dans une base de données compatible SQL, vous devrez les convertir en valeurs nulles. L'exemple suivant effectue cette conversion pour la table EMPLOYES où se trouvent quelques salaires dont le montant est inconnu :

```
UPDATE EMPLOYES
    SET SALAIRE = CASE SALAIRE
                      WHEN -1 THEN NULL
                      ELSE SALAIRE
                  END;
```

La forme abrégée de NULLIF est encore plus simple :

```
UPDATE EMPLOYES
    SET SALAIRE =NULLIF (SALAIRE, -1) ;
```

Un autre cas particulier de CASE : COALESCE

COALESCE, comme NULLIF, est une forme abrégée particulière de l'expression CASE. Elle traite une liste de valeurs qui peuvent être ou ne pas être nulles. Voici comment elle fonctionne :

- **Si une seule des valeurs de la liste n'est pas nulle**, l'expression COALESCE prend cette valeur.
- **Si plus d'une valeur de la liste n'est pas nulle**, l'expression retourne la première valeur non nulle de cette liste.
- **Si toutes les valeurs de la liste sont nulles**, l'expression prend la valeur nulle.

Une expression CASE correspondant à cette démarche prendrait la forme suivante :

```
CASE
    WHEN valeur1 IS NOT NULL
         THEN valeur1
    WHEN valeur2 IS NOT NULL
         THEN valeur2
    ...
    WHEN valeurn IS NOT NULL
        THEN valeurn
    ELSE NULL
END
```

La forme abrégée COALESCE donnera ce qui suit :

```
COALESCE (valeur1, valeur2.... valeurn)
```

Vous utiliserez probablement l'expression COALESCE après avoir effectué une opération OUTERJOIN (elle est abordée au Chapitre 10). Dans ce cas, COALESCE peut vous éviter de saisir une grande quantité de texte.

Conversions de types de données (CAST)

Le Chapitre 2 traite de tous les types de données que SQL reconnaît et supporte. Dans l'idéal, chaque colonne d'une table de base de données doit posséder un type de données précis. Cependant, il n'est pas toujours aussi simple d'effectuer cette attribution. Supposons qu'en définissant une table de votre base de données vous affectiez à une colonne un type de données convenant parfaitement à l'état actuel de votre application. Par la suite, vous voulez étendre le champ de votre application, ou même écrire une application totalement nouvelle qui utilise les données d'une manière différente. Cette nouvelle utilisation peut nécessiter un autre type de données que celui qui a été choisi à l'origine.

Vous pouvez aussi avoir besoin de comparer une colonne d'une table possédant un certain type avec une colonne d'un type différent dans une autre table. Cela se produira par exemple si vous stockez des dates sous la forme de chaînes de caractères dans une table, et encore des dates, mais au format date-heure, dans une seconde table. Même si les deux colonnes contiennent des informations dont la nature est identique (des dates), il n'est normalement pas possible de procéder à la comparaison car leurs types sont différents. Cette incompatibilité pose un problème dans SQL-86 et SQL-89. Cependant, une solution simple existe depuis SQL-92 : l'expression `CAST`.

L'expression `CAST` convertit les données d'une table ou des variables d'un type dans un autre. C'est ce que l'on appelle le transtypage. Après avoir effectué la conversion, vous pouvez effectuer l'opération ou les comparaisons que vous souhaitez.

Bien entendu, l'utilisation de l'expression `CAST` est soumise à certaines restrictions. Vous ne pouvez pas convertir n'importe quel type de données dans un autre. La donnée à convertir doit être compatible avec son nouveau type. Par exemple, vous pouvez utiliser `CAST` pour convertir une chaîne de caractères de type CHAR(10) '2007-04-26' en type `DATE`. Cependant, vous ne pouvez pas utiliser CAST pour convertir la chaîne de caractères CHAR(10) 'rhinocéros' en type DATE. De même, il n'est pas possible `de` transformer un `INTEGER` en un type `SMALLINT` si la valeur de `INTEGER` dépasse la valeur maximale autorisée pour `SMALLINT`.

Vous pouvez convertir une donnée de n'importe quel type caractère dans un autre (tel que numérique ou date), à condition que sa valeur adopte la forme littérale du nouveau type. De même, vous pouvez convertir une donnée d'un type quelconque vers un type caractère à condition que sa valeur prenne la forme littérale du type original.

La liste suivante énumère les problèmes de conversion auxquels vous pouvez être confronté :

- **Un type numérique en n'importe quel type numérique** : si vous convertissez vers un type dont la partie décimale est moins précise, le système arrondira ou tronquera le résultat.

- **Un type numérique exact en un intervalle tel que** `INTERVAL DAY` **ou** `INTERVAL SECOND`.
- **Une** `DATE` **en un** `TIMESTAMP` : la partie heure de `TIMESTAMP` sera remplie de zéros.
- **Un** `TIME` en `TIME` **dont la partie décimale des fractions de seconde est différente, ou en** `TIMESTAMP` : la partie date de `TIMESTAMP` sera remplie avec la date actuelle.
- **Un** `TIMESTAMP` **en** `DATE`, **en** `TIME` **ou en** `TIMESTAMP` **dont la précision de la partie fractionnelle des secondes est différente**.
- **Un** `INTERVAL` **mois-année en un type numérique exact ou en un** `INTERVAL` **année-mois dont la précision du champ préfixe est différente**.
- **Un** `INTERVAL` **jour-heure en un type numérique exact ou en un autre** `INTERVAL` **jour-heure dont la précision du champ préfixe est différente**.

Utiliser CAST avec SQL

Supposez que vous travaillez pour une société qui gère par ordinateur la liste complète de ses employés, qu'ils soient à temps plein ou à temps partiel. Vous stockez la liste des employés à temps plein dans la table EMPLOYES, et vous distinguez chacun d'entre eux par leur numéro de Sécurité sociale que vous stockez sous la forme d'une chaîne du type CHAR(9). Vous dressez la liste des employés à temps partiel dans une table `TEMPS_PARTIEL`. Mais, cette fois, leurs numéros de Sécurité sociale sont enregistrés au format numérique `INTEGER`. Vous allez maintenant générer

la liste des personnes qui apparaissent dans ces deux tables. Vous pouvez faire appel à CAST pour effectuer cette tâche :

```
SELECT * FROM EMPLOYES
    WHERE EMPLOYES.NSS =
      CAST (TEMPS_PARTIEL.NSS AS INTEGER) ;
```

Utiliser CAST avec SQL et un langage hôte

CAST se révèle très utile pour gérer des types de données légaux dans SQL, mais qui n'existent pas dans le langage hôte que vous utilisez. La liste suivante présente quelques exemples de ces types de données :

- Les types DECIMAL et NUMERIC de SQL n'existent pas en Fortran et Pascal.
- Les types FLOAT et REAL de SQL n'existent pas en COBOL standard.
- Le type SQL DATETIME n'existe dans aucun autre langage.

Supposons que vous utilisiez le Fortran ou le Pascal pour accéder à des tables contenant des colonnes au format DECIMAL (5,3). Vous souhaitez éviter les erreurs générées lors de la conversion de ces valeurs dans le type de données REAL du Fortran et du Pascal. Vous pouvez transtyper la donnée dans/ depuis le type chaîne de caractères du langage hôte. Vous récupérez alors un salaire numérique '898,37' sous la forme d'une valeur CHAR(10) '0000898,37'. Pour ce faire, vous allez utiliser CAST afin de transformer le type de données SQL DECIMAL (5,3) en un CHAR(10). Commencez par stocker l'identifiant EMPID d'un employé dans la variable hôte : emp_id_var :

```
SELECT CAST (SALAIRE AS CHAR (10)) INTO
:salaire_var
         FROM EMPLOYES
```

```
            WHERE EMPID = :emp_id_var ;
```

Vous utiliserez ensuite le code suivant pour indiquer à l'application qu'elle doit examiner la valeur chaîne de caractères que contient `salaire_var`, la transformer en une nouvelle valeur `'0000203,74'`, puis la mettre à jour dans la base de données :

```
UPDATE EMPLOYES
    SET SALAIRE = CAST (:salaire_var AS DECIMAL
(5,3))
            WHERE EMPID = :emp_id_var ;
```

Il est difficile de gérer des chaînes de caractères telles que `'000198,37'` en Fortran ou Pascal. Mais vous pouvez écrire un ensemble de sous-programmes qui effectueront les manipulations nécessaires. Il sera ensuite possible de récupérer et mettre à jour n'importe quelle donnée SQL depuis un langage hôte quelconque.

L'idée générale est que `CAST` convient mieux à la conversion de types de données entre un hôte et une base qu'au transtypage à l'intérieur de la base de données elle-même.

Les expressions valeur de ligne

Dans les standards SQL d'origine, tels que SQL-86 et SQL-89, la plupart des opérations traitent une seule valeur ou une seule colonne dans une ligne d'une table. Pour opérer sur plusieurs valeurs, vous devez construire des expressions complexes en utilisant des *connecteurs logiques* (ils sont abordés au Chapitre 10).

Depuis SQL-92, vous pouvez utiliser des expressions *valeur de ligne*. Ces expressions permettent de manipuler une liste de valeurs ou de colonnes, et non plus simplement une seule valeur ou une seule colonne. Elles se présentent sous la forme d'une série d'expressions de valeur séparées par des virgules et délimitées par des parenthèses. Vous pouvez opérer sur une ligne tout entière ou seulement sur une partie de cette ligne.

Le Chapitre 6 explique comment utiliser l'instruction `INSERT` pour ajouter une nouvelle ligne à une table. Pour se faire, l'instruction utilise une

expression valeur de ligne. Par exemple :

```
INSERT INTO ALIMENTS
    (NOM, CALORIES, PROTEINES, GRAISSE,
HYDRATECARBONE)
    VALUES
    ('Fromage, cheddar', 398, 25, 32.2, 2.1)
```

Dans cet exemple, (`'Fromage, cheddar', 398, 25, 32.2, 2.1`) est une expression valeur de ligne. Si vous utilisez de cette manière une telle expression dans une instruction `INSERT`, elle peut contenir des valeurs nulles ou par défaut (une `valeur par défaut` est celle que prend une colonne de la table en l'absence de toute information). La ligne suivante est une expression valeur de ligne autorisée :

```
('Fromage, cheddar', 398, NULL, 32.2, DEFAULT)
```

Vous pouvez ajouter simultanément plusieurs lignes à une table en utilisant de multiples valeurs dans une clause `VALUES`, comme suit :

```
INSERT INTO ALIMENTS
    (NOM, CALORIES, PROTEINES, GRAISSE,
HYDRATECARBONE)
    VALUES
    ('Lait', 14, 1.2, 0.2, 2.5)
('Margarine', 720, 0.6, 81.0, 0.4)
('Moutarde', 75, 4.7, 4.4, 6.4)
('Spaghetti', 148, 5.0, 0.5, 30.1)
```

Il est aussi possible d'utiliser des expressions valeur de ligne pour simplifier l'écriture de comparaisons. Supposez que vous avez deux tables de valeurs nutritionnelles, l'une en français et l'autre en espagnol. Vous recherchez les lignes dans la table en français qui correspondent exactement aux lignes de la table en espagnol. Sans les expressions valeur de ligne, vous devriez écrire quelque chose comme :

```
SELECT * FROM ALIMENTS
```

```
    WHERE ALIMENTS.CALORIES = COMIDA.CALORIA
      AND ALIMENTS.PROTEINES = COMIDA.PROTEINA
      AND ALIMENTS.GRAISSE = COMIDA.GORDO
      AND ALIMENTS.HYDRATECARBO =
COMIDA.CARBOHIDRATO ;
```

Les expressions valeur de ligne vous permettent d'écrire le même code de la manière suivante :

```
SELECT * FROM ALIMENTS
    WHERE (ALIMENTS.CALORIES, ALIMENTS.PROTEINES,
ALI-
MENTS.GRAISSE,
         ALIMENTS.HYDRATECARBO)
      =
         (COMIDA.CALORIA, COMIDA.PROTEINA,
COMIDA.GORDO,
          COMIDA.CARBOHIDRATO) ;
```

Dans cet exemple, le gain de saisie n'est pas énorme. Mais vous pourriez la simplifier davantage si vous compariez plusieurs colonnes. Si le profit n'est que marginal, autant utiliser la syntaxe développée, car elle est plus facile à lire (et donc à corriger ou modifier le cas échéant).

Un autre avantage de l'expression valeur de ligne sur son équivalent codé est qu'elle est plus rapide. En principe, une implémentation « intelligente » devrait être capable d'analyser la version codée et de la transformer en une version valeur de ligne. Mais, en pratique, c'est une opération difficile et délicate qu'aucun SGDB actuel ne sait effectuer.

Chapitre 10
Isoler les données dont vous avez besoin

DANS CE CHAPITRE :

- **Spécifier les tables avec lesquelles vous voulez travailler.**
- **Isoler les lignes dignes d'intérêt.**
- **Construire des clauses `WHERE` efficaces.**
- **Gérer les valeurs nulles.**
- **Construire des expressions composées avec des connecteurs logiques.**
- **Regrouper les résultats d'une requête par colonne.**
- **Ordonner le résultat d'une requête.**
- **Opérer sur des lignes relatives.**

Un système de gestion de base de données a deux fonctions principales : stocker les données et y donner facilement accès. Enregistrer les données n'est pas compliqué. Une armoire à fichiers peut faire la même chose. Toute la difficulté réside dans l'accès aux données. Pour que celles-ci soient utiles, vous devez être capable de séparer le petit nombre d'entre elles qui vous intéresse de l'énorme volume de données dont vous ne voulez pas.

SQL vous permet d'utiliser quelques caractéristiques de la donnée elle-même pour déterminer si une certaine ligne d'une table est d'un quelconque intérêt pour vous. Les instructions `SELECT`, `DELETE` et `UPDATE` indiquent au *moteur* de la base (la partie du SGBD qui agit sur les données) quelle ligne vous allez sélectionner, supprimer ou modifier. Il est possible

d'ajouter des clauses modificatrices aux instructions SELECT, DELETE et UPDATE afin d'affiner la recherche.

Clauses modificatrices

Les clauses modificatrices disponibles en SQL sont FROM, WHERE, HAVING, GROUP BY et ORDER BY. La clause FROM indique au moteur de base de données la ou les tables sur lesquelles vous voulez travailler. Les clauses WHERE et HAVING spécifient une caractéristique de la donnée qui permet de décider s'il faut ou non inclure une ligne particulière dans l'opération courante. Les clauses GROUP BY et ORDER BY précisent comment organiser les lignes récupérées. Le Tableau 10.1 vous présente un récapitulatif.

TABLEAU 10.1 Clauses modificatrices et fonctions.

Clause modificatrice	Fonction
FROM	Spécifie de quelle table les données sont extraites.
WHERE	Filtre les lignes qui ne répondent pas aux critères de recherche.
GROUP BY	Sépare les lignes regroupées selon les valeurs des colonnes de regroupement.
HAVING	Filtre les groupes qui ne correspondent pas aux critères de recherche.
ORDER BY	Trie les résultats produits par les clauses précédentes pour donner le résultat final.

Si vous utilisez simultanément plusieurs de ces clauses, elles doivent apparaître dans l'ordre suivant :

```
SELECT liste_colonnes
    FROM liste_tables
    [WHERE criteres_recherche]
    [GROUP BY colonnes_groupe]
    [HAVING criteres_recherche]
    [ORDER BY criteres_tri]
```

Voici un résumé du rôle de ces clauses :

- La clause `WHERE` est un filtre qui laisse passer les lignes qui correspondent aux critères de recherche et rejette les autres.
- La clause `GROUP BY` ordonne les lignes que la clause `WHERE` laisse passer en fonction des valeurs présentes dans les colonnes de regroupement.
- La clause `HAVING` est un autre filtre qui ne laisse passer que les groupes produits par la clause `GROUP BY` répondant au critère de recherche, les autres étant rejetés.
- La clause `ORDER BY` trie ce qui reste une fois que les clauses précédentes ont traité la table.

Comme la présence de crochets ([]) l'indique, les clauses `WHERE`, `GROUP BY`, `HAVING` et `ORDER BY` sont optionnelles.

SQL évalue ces clauses dans l'ordre `FROM`, `WHERE`, `GROUP BY`, `HAVING` et enfin `SELECT`. Elles sont traitées comme dans un « pipeline », ce qui signifie que chaque clause reçoit le résultat de la clause précédente et produit un résultat pour la clause suivante. En notation fonctionnelle, cet ordre d'évaluation est le suivant :

```
SELECT (HAVING(GROUP BY(WHERE(FROM...))))
```

`ORDER BY` intervient après `SELECT`, ce qui explique pourquoi `ORDER BY` ne peut référencer que des colonnes présentes dans la liste `SELECT`. `ORDER BY` ne peut pas utiliser d'autres colonnes provenant de la ou des tables `FROM`.

Clauses FROM

Il est facile de comprendre la clause FROM si vous n'utilisez qu'une seule table, comme dans l'exemple suivant :

```
SELECT * FROM VENTES ;
```

Cette instruction retourne toutes les données de toutes les lignes de chaque colonne de la table VENTES. Cependant, vous pouvez spécifier plus d'une table dans une clause `FROM`. Considérez l'exemple suivant :

```
SELECT *
    FROM CLIENTS, VENTES ;
```

Cette instruction génère une table virtuelle qui associe les données de la table CLIENTS avec les données de la table VENTES. Chaque ligne de la table CLIENTS est combinée avec chaque ligne de la table VENTES pour former la nouvelle table. Cette dernière contient donc un nombre de lignes égal à celui de la table CLIENTS multiplié par le nombre de lignes de la table VENTES. Si CLIENTS possède dix lignes et que VENTES en a cent, alors la nouvelle table virtuelle en contient mille...

Cette opération est appelée *produit cartésien* de deux tables sources. Le produit cartésien est en fait un type de jointure. Les opérations de *jointure* sont traitées en détail au Chapitre 11.

Dans la plupart des applications, les lignes obtenues à l'issue du produit cartésien de deux tables sont généralement dépourvues de sens. Dans le cas de la table virtuelle formée à partir des tables CLIENTS et VENTES, seules les lignes où le CLIENT_ ID de la table CLIENTS correspond au CLIENT_ID de la table VENTES sont dignes d'intérêt. Vous pouvez éliminer tout le reste en utilisant une clause `WHERE`.

Les clauses WHERE

Dans ce livre, j'ai déjà utilisé de nombreuses fois la clause WHERE sans vraiment la présenter, car sa signification et son utilisation semblent évidentes. Une instruction effectue une opération (comme SELECT, DELETE ou UPDATE) uniquement sur les lignes qui vérifient la condition exprimée par la clause WHERE. La syntaxe de cette clause est la suivante :

```
SELECT liste_colonnes
    FROM nom_table
    WHERE condition ;

DELETE FROM nom_table
    WHERE condition ;
UPDATE nom_table
    SET colonne1=valeur1, colonne2=valeur2,
...co-
lonnen=valeurn
    WHERE condition ;
```

La condition exprimée par la clause WHERE peut être simple ou arbitrairement complexe. À l'aide des connecteurs logiques AND, OR et NOT (ils sont traités plus loin dans ce chapitre), il est possible de combiner plusieurs conditions pour en créer une nouvelle.

Les instructions suivantes vous présentent quelques utilisations de la clause WHERE :

```
WHERE CLIENTS.CLIENT_ID =VENTES.CLIENT_ID
WHERE ALIMENTS.CALORIES = COMIDA.CALORIA
WHERE ALIMENTS.CALORIES < 219
WHERE ALIMENTS.CALORIES > 3 + valeur_base
WHERE ALIMENTS.CALORIES < 219 AND
ALIMENTS.PROTEINES >
27.4
```

Les conditions que la clause `WHERE` exprime sont également appelées prédicats. Un *prédicat* est une expression qui exprime un fait concernant les valeurs.

Par exemple, le prédicat ALIMENTS. CALORIES < 219 est vrai si la valeur de la ligne courante de la colonne ALIMENTS. CALORIES est strictement inférieure à 219. Si cette assertion est vraie, la condition est remplie. Une assertion peut être soit vraie, soit fausse, soit inconnue. Elle est inconnue si un ou plusieurs éléments dans cette assertion sont nuls (c'est-à-dire inconnus, indéfinis, indéterminables, etc.). Les prédicats de comparaison (=, <, >, <>, <= et <=) sont les plus utilisés, mais SQL en comporte d'autres qui vous permettent d'isoler de manière beaucoup plus précise les données que vous recherchez. La liste suivante répertorie les prédicats servant à filtrer les données :

- Prédicats de comparaison
- `BETWEEN`
- `IN [NOT IN]`
- `LIKE [NOT LIKE]`
- `NULL`
- `ALL, SOME, ANY`
- `EXISTS`
- `UNIQUE`
- `OVERLAPS`
- `MATCH`
- `SIMILAR`
- `DISTINCT`

Les prédicats de comparaison

Les exemples de la section précédente sont des usages typiques des prédicats de comparaison : vous y comparez une valeur à une autre. L'opération (`SELECT`, `UPDATE`, `DELETE`, etc.) est appliquée à chaque ligne pour laquelle cette comparaison est évaluée comme étant vraie (`TRUE`), autrement dit pour laquelle la clause `WHERE` est satisfaite. Les lignes pour lesquelles la comparaison renvoie `FALSE` sont éliminées. Considérez par exemple l'instruction SQL suivante :

```
SELECT *FROM ALIMENTS
    WHERE CALORIES < 219 ;
```

Cette instruction affiche toutes les lignes de la table ALIMENTS pour lesquelles la valeur de la colonne CALORIES est inférieure à 219.

Le Tableau 10.2 présente six prédicats de comparaison :

BETWEEN

Il arrive que vous souhaitiez retenir une ligne uniquement si la valeur d'une de ses colonnes est comprise dans une certaine plage. L'utilisation de prédicats de comparaison vous permet d'effectuer ce type de sélection. Par exemple, vous pouvez formuler une clause WHERE servant à sélectionner toutes les lignes de la table ALIMENTS dans lesquelles la valeur de la colonne CALORIES est supérieure à 100 et inférieure à 300 :

TABLEAU 10.2 Prédicats de comparaison SQL.

Comparaison	Symbole
Egal à	=
Différent de	<>
Inférieur à	<
Inférieur ou égal à	<=
Supérieur à	>

Supérieur ou égal à	>=

```
WHERE ALIMENTS.CALORIES > 100 AND
ALIMENTS.CALORIES <300
```

Cette comparaison n'inclut pas les aliments dont la valeur de CALORIES vaut exactement 100 ou 300. Pour inclure ces valeurs limites, vous pouvez écrire l'instruction ci-dessous :

```
WHERE ALIMENTS.CALORIES >=100 AND
ALIMENTS.CALORIES <=300
```

Une autre manière de spécifier une plage qui comprend ces bornes consiste à utiliser le prédicat `BETWEEN` comme suit :

```
WHERE ALIMENTS.CALORIES BETWEEN 100 AND 300
```

Cette clause est fonctionnellement identique à celle de l'exemple précédent, qui utilise des prédicats de comparaison. Comme vous pouvez le constater, cette formulation est plus efficace et un peu plus intuitive que celle qui utilise deux prédicats de comparaison reliés par le connecteur logique `AND`.

Le mot clé `BETWEEN` peut prêter à confusion, car il n'indique pas explicitement si la clause comprend les valeurs limites. En fait, la réponse est oui. De plus, aucun petit oiseau ne va venir se poser sur votre épaule pour vous rappeler que le premier terme de la comparaison *doit* être égal ou inférieur au second. Par exemple, si `ALIMENTS. CALORIES` contient une valeur de 200, la clause suivante est validée :

```
WHERE ALIMENTS.CALORIES BETWEEN 100 AND 300
```

Cependant, la clause ci-dessous, qui semble pourtant équivaloir à l'exemple précédent, fournira le résultat inverse :

```
WHERE ALIMENTS.CALORIES BETWEEN 300 AND 100
```

Si vous utilisez `BETWEEN`, vous devez pouvoir garantir que le premier terme de votre comparaison est toujours égal ou inférieur au second.

Vous pouvez utiliser le prédicat BETWEEN avec les types de données caractère, bit, date/heure ainsi qu'avec les types numériques. Vous pourriez rencontrer quelque chose comme ceci :

```
SELECT PRENOM, NOM
    FROM CLIENTS
    WHERE CLIENTS.NOM BETWEEN 'A' AND 'Mzzz' ;
```

Cet exemple retourne tous les clients dont le nom est dans la première moitié de l'alphabet (à moins que ne commerciez avec M. Mzzzzza).

IN et NOT IN

Les prédicats `IN` et `NOT IN` permettent de vérifier si des valeurs spécifiques (telles que OR, WA et ID) appartiennent à des ensembles particuliers de valeurs (en l'occurrence, des États des États-Unis). Par exemple, une de vos tables contient la liste des fournisseurs américains de consommables que votre société achète régulièrement. Vous voulez connaître les numéros de téléphone des fournisseurs qui se trouvent dans le Nord-Ouest Pacifique. Vous pouvez obtenir ces numéros en utilisant des prédicats de comparaison comme suit :

```
SELECT Societe, Telephone
    FROM FOURNISSEURS
    WHERE Etat = 'OR' OR Etat = 'WA' Or Etat =
'ID' ;
```

Vous pouvez également faire appel au prédicat IN pour réaliser la même tâche :

```
SELECT Societe, Telephone
    FROM FOURNISSEURS
    WHERE Etat IN ('OR', 'WA', 'ID') ;
```

Cette formulation est un peu plus compacte que la précédente. La version `NOT IN` de ce prédicat fonctionne de la même manière. Supposez que vous possédiez des succursales en Californie, en Arizona et au Nouveau-Mexique, et que pour éviter de payer des taxes vous cherchiez à vous

adresser à tous les fournisseurs *sauf* ceux qui résident dans ces États. Utilisez alors la construction suivante :

```
SELECT Societe, Telephone
    FROM FOURNISSEURS
    WHERE Etat NOT IN ('CA', 'AZ', 'NM') ;
```

Utiliser ainsi le mot clé `IN` est un peu plus économique. Ce qui n'est pas forcément en soi un avantage décisif. Vous pourriez tout aussi bien obtenir le même résultat en utilisant les prédicats de comparaison, comme dans le premier exemple de cette section.

Même si l'utilisation de `IN` ne représente pas une économie considérable, elle offre un autre avantage par rapport aux prédicats de comparaison. Votre SGBD implémente probablement ces deux méthodes différemment. De ce fait, l'une d'entre elles peut s'exécuter bien plus rapidement que l'autre sur votre système. Bien entendu, vous allez opter pour la plus rapide. D'ailleurs, un SGBD doté d'un bon optimisateur fera automatiquement le choix de l'efficacité, sans tenir compte de la syntaxe que vous aurez utilisée dans votre requête. Un test de comparaison des performances vous donnera une idée sur la manière de travailler de votre SGBD. Si vous remarquez une différence significative entre les temps d'exécution des deux méthodes, c'est que la qualité de l'optimisateur de votre SGBD peut être remise en question.

Le mot clé `IN` se révèle également utile dans un autre cas. Si `IN` fait partie d'une sous-requête, il vous permet d'extraire des informations provenant de deux tables afin d'obtenir des résultats que vous ne pourriez pas dériver d'une seule table. Le Chapitre 11 revient plus en détail sur les sous-requêtes. Cependant, l'exemple suivant montre comment utiliser le mot clé `IN` dans une sous-requête.

Supposez que vous vouliez afficher tous les noms des clients qui ont acheté le produit F-117A dans les trente derniers jours. Les noms des clients sont stockés dans la table CLIENTS et les transactions commerciales sont mémorisées dans la table TRANSACTIONS. Vous pouvez utiliser la requête suivante :

```
SELECT NOM, PRENOM
    FROM CLIENTS
```

```
WHERE CLIENT_ID IN
  (SELECT CLIENT_ID
     FROM TRANSACTIONS
     WHERE PRODUIT_ID = 'F-117A'
     AND DATE_TRANS >= (DATE_COURANTE -30)) ;
```

Le `SELECT` interne qui porte sur la table TRANSACTIONS est inclus dans le `SELECT` externe qui traite la table CLIENTS. Le `SELECT` interne retourne les numéros CLIENT_ID de tous les clients qui ont acheté le produit F-117A durant les trente derniers jours. Le SELECT externe affiche les noms et prénoms des clients dont le CLIENT_ID a été récupéré par le SELECT interne.

LIKE et NOT LIKE

Vous pouvez utiliser le prédicat `LIKE` pour procéder à une comparaison partielle de deux chaînes de caractère. Les comparaisons partielles sont utiles si vous ne connaissez pas la forme exacte de ce que vous recherchez. Vous pouvez aussi utiliser des comparaisons partielles pour récupérer plusieurs lignes qui contiennent des chaînes similaires dans une colonne donnée.

SQL utilise deux caractères de substitution. Le signe pourcentage(%) remplace n'importe quelle chaîne de caractères qui comprend zéro ou plusieurs caractères. Le trait de soulignement (_) se substitue à un caractère unique quelconque. Le Tableau 10.3 contient quelques exemples d'utilisation de `LIKE`.

Le prédicat `NOT LIKE` retourne toutes les lignes qui ne vérifient pas la comparaison partielle, comme dans :

```
WHERE TELEPHONE NOT LIKE '503%'
```

Cet exemple retourne toutes les lignes de la table dont le numéro de téléphone commence par quelque chose de différent de 503.

TABLEAU 10.3 Le prédicat LIKE de SQL.

Instruction	Valeur retournée
WHERE WORD LIKE ‘intern %’	interne
	international
	internet
WHERE WORD LIKE ‘ % paci %’	océan pacifique
	mouvement pacifique
WHERE WORD LIKE ‘t_p_’	tape
	tapi
	tipi
	type

Il peut arriver que vous recherchiez une chaîne qui contienne elle-même le signe pourcentage ou un trait de soulignement. Dans ce cas, vous voudrez que SQL interprète le signe pourcentage comme le signe « % » et non comme un caractère de substitution. Pour ce faire, vous devrez saisir un caractère *d’échappement* juste avant le signe en question. Vous pouvez utiliser n’im

porte quel caractère comme caractère d’échappement du moment qu’il n’apparaît pas dans la chaîne que vous testez. Par exemple :

```
SELECT CITATION
    FROM BARTLETTS
    WHERE CITATION LIKE ‘20#%’
      ESCAPE ‘#’ ;
```

Le caractère « % » est préfixé du signe d’échappement « # », si bien que l’instruction interprète ce symbole comme le signe « % » et non comme le caractère de substitution. Vous pouvez traiter le trait de soulignement et le caractère d’échappement lui-même de la même manière. Ainsi la requête précédente pourrait renvoyer la citation suivante :

```
20% des commerciaux produisent 80% des résultats
```

Elle retournerait aussi :

```
20%
```

SIMILAR

Le prédicat `SIMILAR` (introduit dans SQL:1999) permet aussi d'effectuer des comparaisons partielles, mais il est plus puissant que `LIKE`. Grâce au prédicat `SIMILAR`, vous pouvez comparer une chaîne de caractères à une expression régulière. Supposez par exemple que vous deviez effectuer une recherche dans la colonne `SYSTEME_EXPLOITATION` d'une table de compatibilité logicielle à la recherche d'une compatibilité avec Microsoft Windows. Vous pourriez construire une clause `WHERE` comme suit :

```
WHERE SYSTEME_EXPLOITATION SIMILAR TO
'('Windows ' (3.1|95|98|Millenium Edi-
tion|CE|NT|2000|XP))' ;
```

Ce prédicat retourne toutes les lignes qui contiennent l'un des systèmes d'exploitation de Microsoft.

NULL

Le prédicat `NULL` retourne toutes les lignes où la valeur de la colonne sélectionnée est nulle. Dans la table ALIMENTS du Chapitre 8, la colonne `HYDRATECARBONE` de plusieurs lignes contient une valeur nulle. Vous pouvez récupérer le nom des aliments correspondants en utilisant :

```
SELECT (ALIMENT)
    FROM ALIMENTS
    WHERE HYDRATECARBONE IS NULL ;
```

Cette requête retourne les valeurs suivantes :

```
Boeuf, hamburger allégé
Poulet, allégé
Opossum, rôti
Porc, jambon
```

Comme vous vous y attendez, le mot clé `NOT` inverse le résultat, comme dans l'exemple suivant :

```
SELECT (ALIMENT)
    FROM ALIMENTS
    WHERE HYDRATECARBONE IS NOT NULL ;
```

Cette requête renvoie toutes les lignes de la table, à l'exception des quatre retournées par l'exemple précédent.

L'instruction `HYDRATECARBONE IS NULL` n'est pas la même que `HYDRATECARBONE = NULL`. Supposez que dans la ligne courante de la table ALIMENTS les champs `HYDRATECARBONE` et `PROTEINES` soient nuls. Vous pourrez en conclure ce qui suit :

- `HYDRATECARBONE IS NULL` est vrai.
- `PROTEINES IS NULL` est vrai.
- `HYDRATECARBONE IS NULL AND PROTEINES IS NULL` est vrai.
- `HYDRATECARBONE = PROTEINES` est inconnu.
- `HYDRATECARBONE = NULL` est une expression illégale.

L'utilisation du mot clé `NULL` dans une comparaison n'a aucun sens, car, dans ce cas, le résultat de la comparaison est toujours *inconnu*.

Pourquoi `HYDRATECARBONE = PROTEINES` est-il inconnu, même si `HYDRATECARBONE` et `PROTEINES` ont la même valeur nulle ? Parce que `NULL` signifie simplement « je ne sais pas ». Si vous ne savez pas à quoi correspondent `HYDRATECARBONE` et `PROTEINES`, vous ne pouvez pas savoir si ces deux valeurs sont identiques. Peut-être que

HYDRATECARBONE vaut 37 et PROTEINES 14, ou peut-être que HYDRATECARBONE et PROTEINES valent chacune 93. Si vous ne savez pas ce que sont ces deux valeurs, vous n'avez aucun moyen de déterminer si elles sont ou non égales.

ALL, SOME, ANY

Il y a des centaines d'années, le philosophe grec Aristote a formulé un système de logique qui est devenu le fondement d'une bonne partie de la pensée occidentale. L'essence de cette logique repose sur l'énoncé de prémisses que vous savez (ou supposez) être vraies, prémisses auxquelles vous appliquez des opérations valides pour arriver à de nouvelles vérités. Voici un exemple de cette procédure :

Prédicat 1 : Tous les Grecs sont humains.

Prédicat 2 : Tous les humains sont mortels.

Conclusion : Tous les Grecs sont mortels.

Autre exemple :

Prédicat 1 : Certains Grecs sont des femmes.

Prédicat 2 : Toutes les femmes sont des êtres humains.

Conclusion : Certains Grecs sont des êtres humains.

Le même raisonnement logique pourrait être formulé d'une manière un peu différente :

Si quelques Grecs sont des femmes, et que toutes les femmes sont des êtres humains, alors quelques Grecs sont des êtres humains.

Le premier exemple utilise le quantificateur universel ALL (*tous*) dans les deux prémisses, ce qui vous permet de procéder à une déduction générale sur les Grecs dans la conclusion. Le deuxième exemple utilise le

quantificateur existentiel SOME (*certains, quelques*) dans la première prémisse, ce qui vous permet de procéder à une déduction sur un certain nombre de Grecs dans la conclusion. Le troisième exemple (qui s'écrirait en anglais *if any Greeks...)* utilise un quantificateur existentiel (ANY) qui est un synonyme de SOME pour arriver à la même conclusion que le deuxième exemple. Voyons comment ANY et SOME sont traités en SQL en passant directement de la Grèce aux États-Unis.

Prenons l'exemple de statistiques de base-ball. Le base-ball est un sport très éprouvant, en particulier pour les lanceurs. Un lanceur doit lancer la balle entre 90 et 150 fois lors d'une partie, ce qui est très fatigant. C'est pourquoi le lanceur est souvent remplacé par un autre joueur avant la fin de la partie. Tenir le rôle de lanceur pendant toute une partie est un véritable exploit, quel que soit le résultat du match.

Supposez que vous conserviez toutes les grandes parties dans lesquelles tous les grands lanceurs ont joué. Dans une table, vous allez dresser la liste des lanceurs de la Ligue américaine, et dans une autre table celle des lanceurs de la Ligue nationale. Chaque table contient pour un joueur son nom, son prénom et le nombre de matches entiers auxquels il a participé.

Supposons maintenant que vous souteniez la théorie selon laquelle les lanceurs de la Ligue américaine jouent des parties entières plus souvent que ceux de la Ligue nationale. Pour vérifier cette hypothèse, vous allez formuler la requête suivante :

```
SELECT NOM, PRENOM
    FROM LIGUE_AMERICAINE
    WHERE PARTIES_COMPLETES > ALL
      (SELECT PARTIES_COMPLETES
         FROM LIGUE_NATIONALE) ;
```

La sous-requête (SELECT interne) retourne une liste qui indique le nombre de parties complètes auxquelles a participé chaque lanceur de la Ligue nationale. La requête externe retourne ensuite le nom et le prénom des lanceurs de la Ligue américaine qui ont participé à plus de parties complètes que tous (ALL) les lanceurs de la Ligue nationale.

Regardez maintenant l'instruction suivante :

```
SELECT NOM, PRENOM
    FROM LIGUE_AMERICAINE
    WHERE PARTIES_COMPLETES > ANY
      (SELECT PARTIES_COMPLETES
         FROM LIGUE_NATIONALE) ;
```

Dans ce cas, vous utilisez le quantificateur existentiel ANY, et non le quantificateur universel ALL. La sous-requête (interne) est identique à celle de l'exemple précédent. Elle retourne à nouveau la liste de toutes les parties complètes auxquelles ont participé les lanceurs de la Ligue nationale. La requête externe retourne le nom et le prénom de tous les lanceurs de la Ligue américaine qui ont participé à plus de matches entiers que n'importe quel lanceur de la Ligue nationale. Comme vous pouvez être virtuellement certain qu'au moins un des lanceurs de la Ligue nationale n'a pas participé à un match entier, le résultat comprend probablement tous les lanceurs de la Ligue américaine qui ont réalisé au moins une partie complète.

Si vous remplacez le mot clé ANY par le mot clé équivalent SOME, le résultat est identique. Si l'assertion « au moins un lanceur de la Ligue nationale n'a pas participé à une partie complète » est vérifiée, il devient exact de dire « quelques lanceurs de la Ligue nationale n'ont pas participé à une partie complète ».

EXISTS

Vous pouvez utiliser le prédicat EXISTS avec une sous-requête pour déterminer si cette dernière retourne ou non des lignes. Si la sous-requête retourne au moins une ligne, c'est que le résultat vérifie la condition EXISTS. La requête externe est alors exécutée. Par exemple :

```
SELECT NOM, PRENOM
    FROM CLIENTS
    WHERE EXISTS
      (SELECT DISTINCT CLIENT_ID
         FROM VENTES
         WHERE VENTES.CLIENT_ID =
```

```
CLIENTS.CLIENT_ID) ;
```

La table VENTES contient un enregistrement de toutes les ventes de la société. On y trouve le `CLIENT_ID` de chaque client ayant effectué un achat ainsi que d'autres informations pertinentes. La table CLIENTS contient le nom et le prénom de chaque client, mais aucune information sur des transactions spécifiques.

La sous-requête de l'exemple précédent renvoie une ligne pour chaque client qui a effectué au moins un achat. La requête externe retourne le nom et le prénom des clients qui ont effectué des achats référencés dans la table VENTES.

`EXISTS` est l'équivalent d'une comparaison de `COUNT` avec zéro, comme le montre la requête suivante :

```
SELECT NOM, PRENOM
    FROM CLIENTS
    WHERE 0 <>
    (SELECT COUNT (*)
  FROM VENTES
WHERE VENTES.CLIENT_ID =CLIENTS.CLIENT_ID) ;
```

Pour chaque ligne qui contient un `CLIENT_ID` égal à un `CLIENT_ ID` de la table CLIENTS, cette instruction renvoie les colonnes NOM et PRENOM de la table CLIENTS. En d'autres termes, elle affiche le nom du client qui a effectué l'achat pour chaque transaction de la table VENTES.

UNIQUE

Tout comme pour `EXISTS`, vous pouvez utiliser le prédicat `UNIQUE` avec une sous-requête. Mais alors `qu'EXISTS` est évalué comme étant vrai si et seulement si la sous-requête renvoie au moins une ligne, le prédicat `UNIQUE` est validé si et seulement si la sous-requête ne retourne pas deux lignes identiques. Autrement dit, `UNIQUE` n'est vrai que quand toutes les lignes fournies par la sous-requête sont uniques.

Prenons l'exemple suivant :

```
SELECT NOM, PRENOM
    FROM CLIENTS
    WHERE UNIQUE
      (SELECT CLIENT_ID FROM VENTES
      WHERE VENTES.CLIENT_ID = CLIENTS.CLIENT_ID)
;
```

Cette instruction retrouve les noms de tous les clients pour lesquels la table VENTES ne recense qu'une seule transaction. Notez bien que deux valeurs nulles ne sont pas considérées comme étant égales et sont donc uniques. Lorsque le mot clé `UNIQUE` est appliqué à une table de résultat qui ne contient que deux lignes nulles, le prédicat `UNIQUE` est évalué comme ayant la valeur `True` (vrai).

DISTINCT

Le prédicat `DISTINCT` est comparable à `UNIQUE`, si ce n'est qu'il ne traite pas les valeurs nulles de la même manière. Si toutes les valeurs de la table résultat sont `UNIQUE`, alors elles sont `DISTINCT` les unes des autres. Cependant, le prédicat `DISTINCT` n'est pas vérifié quand il est appliqué à une table résultat qui ne contient que deux lignes nulles. Deux valeurs nulles ne sont pas considérées comme étant distinctes, tout en étant quand même uniques.

Cela peut sembler paradoxal, mais il n'y a aucune raison à cela. Dans certains cas, vous voulez traiter deux valeurs nulles comme étant différentes l'une de l'autre. Dans d'autres circonstances, vous avez besoin de les placer dans un même sac, et donc de les considérer comme identiques. Avec la première situation, vous utiliserez le prédicat `UNIQUE` ; dans la seconde, vous ferez appel au prédicat `DISTINCT`.

OVERLAPS

Le prédicat `OVERLAPS` permet de déterminer si deux intervalles de temps se chevauchent. Il est utile pour éviter par exemple des conflits d'emploi du

temps. Si deux intervalles se chevauchent, le prédicat est évalué comme étant vrai. Sinon, il n'est pas validé.

Vous pouvez spécifier un intervalle de deux manières : soit à l'aide d'un point de départ et d'un point final, soit à partir d'un point de départ et d'une durée. Voici quelques exemples :

```
(TIME '2:55:00', INTERVAL '1' HOUR)
OVERLAPS
(TIME '3:30:00', INTERVAL '2' HOUR)
```

L'exemple précédent renvoie True, car 3 heures 30 est moins d'une heure après 2 heures 55.

```
(TIME '9:00:00', TIME '9:30:00')
OVERLAPS
(TIME '9:29:00', TIME '9:31:00')
```

L'exemple précédent renvoie True, car les deux intervalles se chevauchent d'une minute.

```
(TIME '9:00:00', TIME '10:00:00')
OVERLAPS
(TIME '10:15:00', INTERVAL '3' HOUR)
```

L'exemple précédent renvoie `False`, car les deux intervalles ne se recouvrent pas.

```
(TIME '9:00:00', TIME '9:30:00')
OVERLAPS
(TIME '9:30:00', TIME '9:35:00')
```

Cet exemple renvoie `False`, car même si les deux intervalles sont contigus, ils ne se chevauchent pas.

MATCH

Le respect de l'intégrité référentielle, traitée au Chapitre 5, implique de maintenir la cohérence d'une base de données qui contient plusieurs tables. Vous pouvez perdre cette intégrité en ajoutant à une table enfant une ligne sans correspondance dans la table parent. Des problèmes similaires se posent si vous supprimez une ligne de la table parent alors qu'une ligne correspondante existe dans la table enfant.

Supposons à nouveau que vous ayez une table CLIENTS qui conserve la trace de tous vos clients, et une table VENTES qui enregistre toutes les transactions. Vous ne voulez évidemment pas ajouter une ligne à VENTES tant que vous n'avez pas saisi dans la table CLIENTS les références du client qui a effectué un achat. De même, vous ne voulez pas supprimer un client s'il a effectué un ou plusieurs achats mémorisés dans la table VENTES. Avant d'effectuer une opération d'insertion ou de suppression, vous devrez donc vérifier qu'elle n'engendre pas des problèmes d'intégrité. Le prédicat `MATCH` peut effectuer un tel contrôle.

Dans notre exemple, `CLIENT_ID` est la clé primaire de la table CLIENTS et elle joue le rôle d'une clé étrangère dans la table VENTES. Chaque ligne de la table CLIENTS doit comporter un `CLIENT_ID` unique et non nul. Par contre, `CLIENT_ID` n'est pas unique dans la table VENTES, car un client peut effectuer plusieurs achats. Cette situation ne pose pas de problème et ne compromet pas l'intégrité, car `CLIENT_ID` est ici une clé étrangère et non une clé primaire.

De même, `CLIENT_ID` peut être nul dans la table VENTES, car quelqu'un peut entrer dans votre magasin, acheter quelque chose et partir avant même que vous ayez eu le temps de saisir son nom et son adresse dans la table CLIENTS. Cette situation peut créer une ligne dans la table enfant qui n'a pas de correspondance dans la table parent. Pour contourner ce problème, il suffit de créer un client générique dans la table CLIENTS et de lui attribuer tous les achats anonymes.

Supposez qu'un client entre dans votre magasin et prétende avoir acheté un F-117A Stealth Fighter le 18 mai 2006. Il a perdu son ticket de caisse, mais il veut à tout prix vous rendre l'avion, car il est visible sur les radars ennemis. Vous pouvez vérifier ses propos en recherchant une correspondance dans votre base de données. En premier lieu, vous allez récupérer son `CLIENT_ID` dans la variable `vclientid`, puis vous utiliserez la syntaxe suivante :

```
...WHERE (:vclientid, 'F-117A', 2006-05-18')
         MATCH
    (SELECT CLIENT_ID, PRODUIT_ID, DATE_VENTE
         FROM VENTES)
```

Si une vente de F-117A existe pour ce client à cette date, le prédicat MATCH va renvoyer la valeur `True`. Vous allez devoir reprendre l'appareil défectueux et rembourser le client. (**Note** : Si une des valeurs du premier argument de `MATCH` est nulle, le prédicat est toujours vérifié.)

Les développeurs de SQL ont ajouté les prédicats `MATCH` et `UNIQUE` pour la même raison : ils permettent d'effectuer des tests explicites afin de vérifier l'intégrité référentielle implicite et les contraintes `UNIQUE`.

La forme générale du prédicat `MATCH` est la suivante :

```
Valeur_ligne MATCH [UNIQUE] [SIMPLE] PARTIAL /
FULL]
Sous_requête
```

Les options `UNIQUE`, `PARTIAL` et `FULL` interviennent quand `Valeur_ligne` désigne une ligne dont une ou plusieurs colonnes sont nulles (voir le Chapitre 9 pour plus d'informations sur les expressions de valeur de ligne). Les règles régissant le prédicat `MATCH` sont une copie des règles d'intégrité référentielle correspondantes.

Les règles d'intégrité référentielle et le prédicat MATCH

Les *règles d'intégrité référentielle* imposent que les valeurs d'une colonne dans une table correspondent à celles d'une ou aux colonnes de la première table comme la *clé étrangère*, et aux colonnes de la seconde table comme étant la *clé primaire* ou *clé unique*. Par exemple, vous pouvez déclarer la colonne `EMP_NODPT` de la table EMPLOYE comme une clé étrangère qui référence la colonne `NODEPT` dans la table DEPT. Cela vous permet de vous assurer que si vous enregistrez un employé dans la table EMPLOYE

comme travaillant au département 123, une ligne dont la valeur NODEPT est égale à 123 apparaîtra dans la table DEPT.

Si la clé étrangère et la clé primaire ne concernent qu'une seule colonne, la situation est assez simple. Mais les deux clés sont souvent formées à partir de plusieurs colonnes. Par exemple, la valeur NODEPT peut n'être unique que pour un seul endroit, si bien que pour identifier parfaitement une colonne DEPT, vous devez spécifier à la fois un LIEU et un NODEPT. Si les bureaux de Boston et de Tampa ont tous deux un département 123, ils correspondront à ('Boston', '123') et ('Tampa', '123'). Dans ce cas, la table EMPLOYE doit posséder deux colonnes pour identifier un DEPT. Nommez ces colonnes LIEU_EMP et NODEPT_EMP. Si un employé travaille au département 123 à Boston, LIEU_EMP et NO-DEPT_EMP vaudront respectivement 'Boston' et '123'. La déclaration étrangère de l'employé est alors la suivante :

```
FOREIGN KEY (LIEU_EMP, NODEPT_EMP)
    REFERENCES DEPT (LIEU, NODEPT)
```

Tirer des conclusions valides de l'étude de vos données peut devenir extrêmement problématique si elles contiennent des valeurs nulles. Vous allez parfois traiter des données qui contiennent des valeurs nulles d'une certaine façon, et dans d'autres cas vous les traiterez d'une autre manière. Les mots clés UNIQUE, SIMPLE, PARTIAL et FULL sont autant de manières de travailler sur des données qui contiennent des valeurs nulles. Si vous n'êtes pas concerné par cette question, sautez le reste de cette section et passez directement à la suivante, « Connecteurs logiques ». Sinon, lisez attentivement les paragraphes suivants. Chaque entrée de la liste présente un cas particulier de valeurs nulles et explique comment le prédicat MATCH les gère.

Voici les scénarios qui illustrent les règles régissant les valeurs nulles et le prédicat MATCH :

- **Les deux valeurs sont de même nature** : Si les valeurs de LIEU_EMP et de NODEPT sont toutes deux non nulles ou toutes deux nulles, les règles d'intégrité référentielle

sont les mêmes que pour des clés basées sur une colonne unique dont les valeurs sont nulles ou non nulles.

- **Une valeur est nulle et l'autre pas :** Si, par exemple, `LIEU_EMP` est nulle et que `NODEPT` est non nulle, ou si `LIEU_ EMP` est non nulle et `NODEPT` est nulle, vous avez besoin de nouvelles règles. Quelle règle appliquer si vous mettez à jour la table EMPLOYE avec les valeurs de `LIEU_EMP` et de `NODEPT` de (`NULL '123'`) ou (`'Boston'`, `NULL`) ? Six alternatives se présentent : `SIMPLE`, `PARTIAL` et `FULL`, chacune combinée ou non au mot clé `UNIQUE`.
- **Le mot clé `UNIQUE` est présent** : Il spécifie qu'une colonne correspondante dans la table résultat de la sous-requête doit être unique pour que le prédicat soit validé.
- **Les deux composants de la valeur de R sont nuls** : Le prédicat `MATCH` est validé quel que soit le contenu de la table résultat de la sous-requête à comparer.
- **Aucun des composants de la valeur de ligne R n'est nul,** `SIMPLE` **est spécifié**, `UNIQUE` **n'est pas spécifié et au moins une ligne de la table sous-requête correspond à R** : le prédicat `MATCH` renvoie la valeur `True`. Sinon, il retourne `False`.
- **Aucun des composants de la valeur de ligne R n'est nul,** `SIMPLE` **est spécifié**, `UNIQUE` **est spécifié et au moins une ligne de la table résultat de la sous-requête est à la fois** `UNIQUE` **et correspond à R :** le prédicat `MATCH` renvoie la valeur `True`. Sinon, il retourne `False`.

» **Un composant de l'expression valeur de ligne R est nul et** `SIMPLE` **est spécifié** : Le prédicat `MATCH` est vérifié.

UNE RÈGLE DÉMOCRATIQUE

La version SQL-89 du standard spécifiait que la règle UNIQUE était la règle par défaut sans que quiconque ait débattu de possibles alternatives. Au cours du développement de la version SQL-92 du standard, plusieurs propositions furent suggérées. Certaines personnes préféraient de loin les règles `PARTIAL` et demandaient à ce qu'elles soient les uniques règles. Elles estimaient que les règles `UNIQUE` de SQL-89 posaient d'énormes problèmes et souhaitaient qu'elles soient considérées comme un bogue et que les règles `PARTIAL` soient jugées comme étant des corrections.

D'autres personnes penchaient pour les règles `UNIQUE`, jugeant les règles `PARTIAL` obscures, sujettes à erreurs et inefficaces. D'autres encore souhaitaient s'en tenir à la discipline supplémentaire offerte par les règles `FULL`. Pour finir, les trois mots clés furent intégrés au standard, si bien que l'utilisateur peut choisir celui qu'il préfère. SQL:1999 a ajouté à tout cela les règles `SIMPLE`. Cette prolifération rend la gestion des valeurs nulles tout sauf simple. Si `SIMPLE`, `PARTIAL` ou `FULL` ne sont pas spécifiées, les règles `SIMPLE` sont appliquées.

» **Un composant quelconque de l'expression valeur de ligne R n'est pas nul,** `PARTIAL` **est spécifié,** `UNIQUE` **n'est pas spécifié, et les parties non nulles d'au moins une ligne de la table résultat produite par la sous-**

requête correspondent à R : Le prédicat `MATCH` renvoie la valeur `True`. Sinon, il retourne `False`.

- **Un composant quelconque de l'expression valeur de ligne R n'est pas nul,** `PARTIAL` **est spécifié,** `UNIQUE` **est spécifié, et les parties non nulles de R correspondent aux parties non nulles d'au moins une ligne unique de la table résultat de la sous-requête** : Le prédicat `MATCH` renvoie la valeur `True`. Sinon, il retourne `False`.
- **Aucun des composants de la valeur de ligne R n'est nul,** `FULL` **est spécifié,** `UNIQUE` **n'est pas spécifié, et au moins une ligne de la table résultat de la sous-requête correspond à R :** Le prédicat `MATCH` renvoie la valeur `True`. Sinon, il retourne `False`.
- **Aucun des composants de la valeur de ligne R n'est nul,** `FULL` **est spécifié,** `UNIQUE` **est spécifié, et au moins une ligne de la table résultat de la sous-requête est à la fois** `UNIQUE` **et correspond à R :** Le prédicat `MATCH` renvoie la valeur `True`. Sinon, il retourne `False`.
- **Un composant quelconque de l'expression valeur de ligne R est nul, et** `FULL` **est spécifié :** Le prédicat `MATCH` renvoie la valeur `True`. Sinon, il retourne `False`.

Connecteurs logiques

Comme les exemples précédents l'ont démontré, une seule condition n'est quelquefois pas suffisante pour récupérer toutes les lignes que vous voulez extraire d'une table. Dans certains cas, ces lignes doivent remplir une ou plusieurs conditions. Dans d'autres situations, cela suffit simplement à les qualifier pour la partie suivante. Dans d'autres situations encore, vous

voudrez peut-être récupérer uniquement les lignes qui ne remplissent pas certaines conditions. Pour répondre à tous ces besoins, SQL propose les connecteurs logiques AND, OR et NOT.

AND

Si l'obtention d'une ligne est soumise à la validité conjointe de plusieurs conditions, utilisez le connecteur logique AND. Par exemple :

```
SELECT NO_FACTURE, DATE_VENTE, VENDEUR,
TOTAL_VENTE
    FROM VENTES
    WHERE DATE_VENTE > =‘2006-05-15’
      AND DATE_VENTE <= ‘2006-05-21’ ;
```

La clause WHERE doit remplir les deux conditions suivantes :

- DATE_VENTE doit être supérieur ou égal au 15 mai 2006.
- DATE_VENTE doit être inférieur ou égal au 21 mai 2006.

Seules les lignes qui correspondent à des ventes effectuées dans la semaine du 15 mai 2006 remplissent les deux conditions. La requête ne retourne que ces lignes.

Remarquez que le connecteur AND est strictement logique. Cette restriction peut prêter quelquefois à confusion, car certaines personnes emploient le mot « et » (AND) dans un contexte moins précis. Par exemple, votre patron vous dit : « Je voudrais voir les résultats des ventes de Dupont et Nemours. » Il a dit Dupont et Nemours ; par conséquent, vous allez certainement écrire la requête SQL suivante :

```
SELECT *
    FROM VENTES
    WHERE VENDEUR = ‘Dupont’
      AND VENDEUR =‘Nemours’ ;
```

Mais en fait votre patron avait une autre idée en tête :

```
SELECT *
    FROM VENTES
    WHERE VENDEUR IN (‘Dupont’, ‘Nemours’) ;
```

La première requête ne retournera rien, car aucune des ventes de la table VENTES n’a été effectuée à la fois par Dupont et Nemours. La seconde requête renverra les informations concernant les ventes effectuées par Dupont ou par Nemours, et c’est ce que votre patron demandait probablement.

OR

Si une ou plusieurs conditions doivent être validées pour qu’une ligne soit retournée comme résultat, utilisez le connecteur logique `OR`. Par exemple :

```
SELECT NO_FACTURE, DATE_VENTE, VENDEUR,
TOTAL_VENTE
    FROM VENDEUR
      WHERE VENDEUR = ‘Dupont’
         OR TOTAL_VENTE > 2000 ;
```

Cette requête retourne toutes les ventes faites par Dupont, quelle que soit leur importance, ainsi que toutes les ventes de plus de 2 000 euros quelle que soit la personne qui les a effectuées.

NOT

Le connecteur de négation `NOT` inverse le sens d’une condition. Si une condition doit normalement retourner une valeur vraie, ajouter `NOT` lui fait renvoyer une valeur fausse. Si une condition renvoie normalement une valeur fausse, ajouter `NOT` lui fait retourner une valeur vraie. Prenons l’exemple suivant :

```
SELECT NO_FACTURE, DATE_VENTE, VENDEUR
    FROM VENTES
      WHERE NOT (VENDEUR = ‘Dupont’) ;
```

Cette requête retourne les lignes de toutes les ventes que les vendeurs autres que Dupont ont effectuées.

Lorsque vous utilisez `AND`, `OR` et `NOT`, il peut arriver que la portée du connecteur ne soit pas claire. Utilisez des parenthèses pour vous assurer que SQL l'applique au prédicat voulu. Dans l'exemple précédent, le connecteur `NOT` s'applique au prédicat (`VENDEUR= 'Dupont'`).

Clauses GROUP BY

Si vous récupérez des lignes d'une table en utilisant l'instruction `SELECT`, ces lignes vous sont retournées par défaut dans l'ordre où elles apparaissent dans la table source. Cet ordre n'a quelquefois aucun sens. La clause `GROUP BY` vous permet alors de spécifier une ou plusieurs colonnes qui déterminent l'appartenance d'une entrée à un groupe. Précisons cela.

Supposez que vous soyez un responsable commercial et que vous souhaitiez examiner les performances de votre force de vente pour une semaine donnée. Les informations produites doivent être groupées par vendeur. Vous pourriez simplement écrire l'instruction suivante :

```
SELECT NO_FACTURE, DATE_VENTE, VENDEUR,
TOTAL_VENTE
    FROM VENTES ;
```

La réponse se présenterait alors comme à la Figure 10.1.

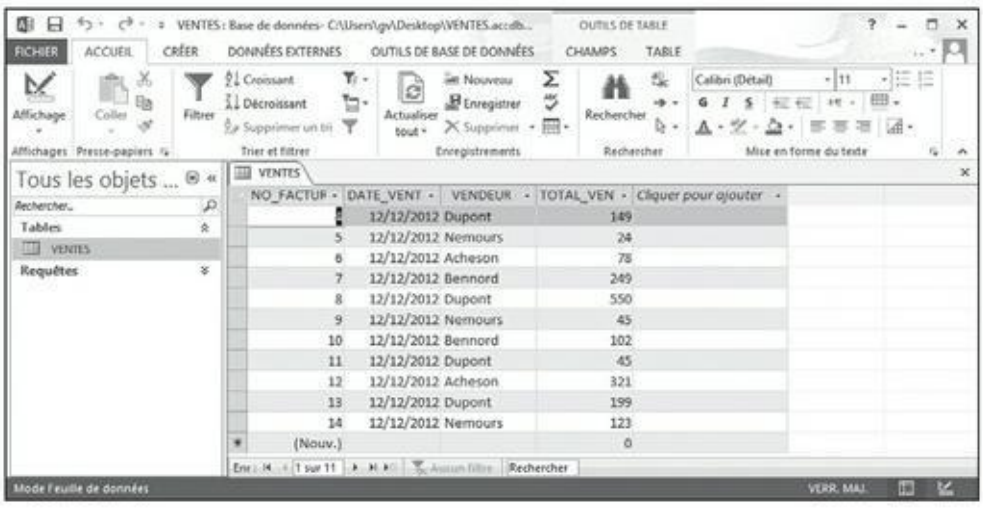


FIGURE 10.1 Résultat des ventes d'une période donnée.

Comme il n'y a que peu de ventes, ce résultat vous donne déjà une certaine idée du travail de vos représentants. Evidemment, dans la vraie vie, il y aurait beaucoup plus de ventes (heureusement !), et il serait difficile de dire si vos objectifs ont été atteints. Pour obtenir l'analyse dont vous avez besoin, vous pouvez combiner une clause `GROUP BY` avec des fonctions d'agrégation (ou d'ensemble) afin d'afficher une vision quantitative des performances du département commercial. L'instruction suivante classe les vendeurs selon la moyenne de leur chiffre d'affaires :

```
SELECT VENDEUR, AVG(VENTE_TOTAL)
    FROM VENTES
    GROUP BY VENDEUR ;
```

Le résultat de cette requête, exécutée dans Microsoft Access 2013, est illustré à la Figure 10.2. Vous pouvez l'exécuter avec un autre gestionnaire de base de données, mais le résultat affiché sera peut-être différent :

À l'évidence, la moyenne des ventes de Dupont est considérablement supérieure à celle de ses collègues. Vous pouvez vérifier la véracité de ce fait en demandant le total des ventes réalisées par chaque commercial :

```
SELECT VENDEUR, SUM(VENTE_TOTAL)
    FROM VENTES
    GROUP BY VENDEUR ;
```

Ce qui affichera le résultat reproduit à la Figure 10.3 qui est cohérent avec les informations précédentes.

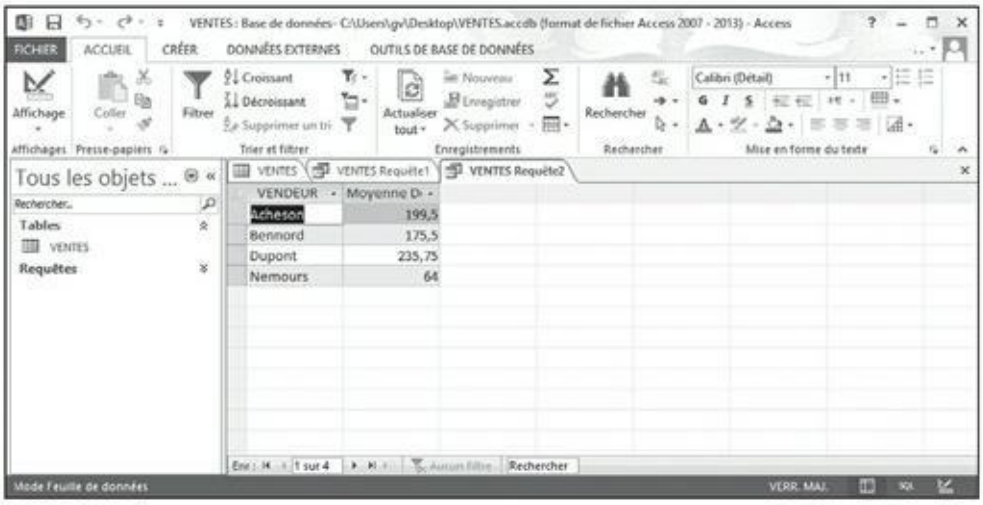


FIGURE 10.2 La moyenne des ventes de chaque vendeur.

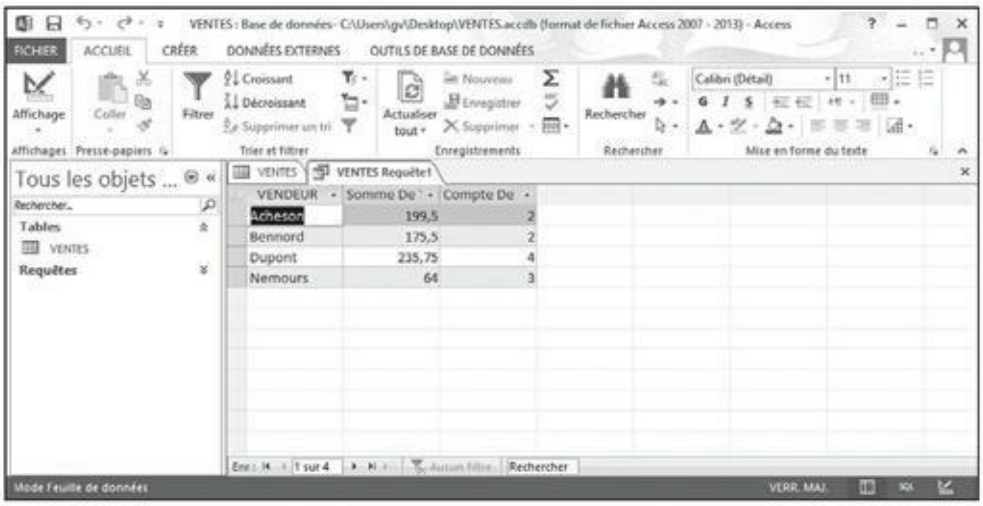


FIGURE 10.3 Total des ventes de chaque vendeur.

Les clauses HAVING

Une clause `HAVING` est un filtre qui applique une restriction sur la table virtuelle qu'une clause GROUP BY vient de créer. Tout comme une clause WHERE exclut des lignes d'une requête, une clause HAVING exclut des groupes.

Reprenons l'exemple de la section précédente. Supposez que le responsable commercial veuille se concentrer sur les performances des vendeurs autres que celles de Dupont (vu ses résultats, il doit constituer une classe à lui tout

seul). Vous pouvez le lui permettre en ajoutant une clause HAVING à la requête précédente, comme suit :

```
SELECT VENDEUR, SUM(VENTE_TOTAL)
FROM VENTES
GROUP BY VENDEUR ;
HAVING VENDEUR <> 'Dupont' ;
```

Le résultat sera le suivant :

```
VENDEUR Total
------- -----------
Nemours 12.48
Podoleck 12.00
```

Les clauses ORDER BY

Utilisez la clause ORDER BY pour afficher une table résultat dans un ordre alphabétique croissant ou décroissant. Alors que la clause GROUP BY réunit les lignes par groupes classés par ordre alphabétique, la clause ORDER BY trie pour sa part des lignes individuelles. Elle doit être la dernière clause que vous spécifiez dans une requête. Si cette requête contient également la clause GROUP BY, celle-ci provoque d'abord un regroupement des lignes, puis la clause ORDER BY trie ces lignes à l'intérieur de chacun des groupes qui auront été formés. Si vous n'utilisez pas la clause GROUP BY, l'instruction considère la table tout entière comme un groupe. ORDER BY trie alors toutes ces lignes en fonction de la colonne ou des colonnes que vous aurez spécifiées.

Pour illustrer cela, considérez les données de la table VENTES. Elle contient des colonnes pour NO_FACTURE, DATE_VENTE, VENDEUR et TOTAL_VENTE. Si vous utilisez l'exemple suivant, vous pourrez visualiser les données VENTES selon un ordre arbitraire :

```
SELECT * FROM VENTES ;
```

Dans une certaine implémentation, l'ordre du tri peut être celui dans lequel vous avez inséré des lignes dans la table. Dans d'autres cas, le classement sera celui de la dernière mise à jour. Et si quelqu'un a physiquement réorganisé la base de données, le résultat sera pratiquement imprévisible. En règle générale, vous devez spécifier l'ordre dans lequel vous voulez visualiser les données. Vous pouvez par exemple trier les lignes en fonction de la date de vente (DATE_VENTE) de la manière suivante :

```
SELECT * FROM VENTES ORDER BY DATE_VENTE ;
```

Cet exemple retourne toutes les lignes de la table VENTES groupées en fonction de DATE_VENTE.

Les lignes dont les colonnes DATE_VENTE sont identiques seront triées entre elles selon un ordre par défaut qui dépend de l'implémentation. Cependant, vous pouvez spécifier vous-même comment trier les lignes en fonction de la date de vente, par exemple dans l'ordre des numéros de factures :

```
SELECT * FROM VENTES ORDER BY DATE_VENTE,
NO_FACTURE ;
```

Cette requête trie tout d'abord les ventes par DATE_VENTE. Puis, pour chaque date de vente, elle ordonne les résultats par NO_ FACTURE. Attention ! Ne confondez pas cet exemple avec la requête suivante :

```
SELECT* FROM VENTES ORDER BY NO_FACTURE,
DATE_VENTE ;
```

Cette requête trie d'abord les ventes par NO_FACTURE, puis par DATE_VENTE pour chaque NO_FACTURE différent. Ce qui ne produira probablement pas le résultat escompté, car il est peu vraisemblable qu'une seule facture comporte plusieurs dates.

La requête suivante est un autre exemple de la manière dont SQL peut retourner les données :

```
SELECT* FROM VENTES ORDER BY VENDEUR, DATE_VENTE
;
```

Cet exemple trie par vendeur, puis par date de vente. Une fois les données visualisées dans cet ordre, vous pouvez l'inverser de la manière suivante :

```
SELECT * FROM VENTES ORDER BY DATE_VENTE, VENDEUR
;
```

Cet exemple trie les lignes selon DATE_VENTE, puis selon la colonne VENDEUR.

Tous ces exemples de tri se font par défaut en ordre croissant (ASC). Le dernier SELECT classe tout d'abord les VENTES en commençant par les plus récentes, puis, pour une date donnée, affiche les VENTES pour 'Albert' avant 'Bernard'. Si vous préférez un ordre décroissant (DESC), vous pouvez le spécifier pour une ou plusieurs colonnes de tri de la manière suivante :

```
SELECT *FROM VENTES
ORDER BY DATE_VENTE DESC, VENDEUR ASC ;
```

Cet exemple spécifie un ordre décroissant pour les dates de vente (les plus récentes étant donc affichées en premier), puis un ordre croissant pour les vendeurs, ce qui les classe par ordre alphabétique.

Les limites de Fetch

Quand on a changé le standard ISO/IEC SQL, on a développé ses capacités de langage, ce qui est en soi une bonne chose. Mais il arrive quelquefois que de tels changements ne permettent pas d'anticiper toutes les conséquences possibles. C'est ce qui s'est produit avec les limitations de FETCH dans SQL:2008.

L'idée de limiter FETCH est due au fait qu'une instruction SELECT peut retourner un nombre indéterminé de lignes, alors que quelquefois seules les trois ou les dix premières lignes sont intéressantes. C'est dans cet esprit que l'on a ajouté dans SQL: : 2008 la syntaxe suivante :

```
SELECT Vendeur, AVG(TOTAL_VENTE)
FROM VENTES
```

```
GROUP BY Vendeur
ORDER BY AVG(TOTAL_VENTE) DESC
FETCH FIRST 3 ROWS ONLY;
```

Tout va bien. Vous obtenez ainsi les noms des trois premiers vendeurs qui ont vendu des produits très chers. Mais il reste un petit problème : que faire si ces trois vendeurs ont le même total de ventes ? Seul l'un de ces trois noms sera retourné. Lequel ? C'est indéterminé.

L'indétermination n'est pas acceptable pour toute personne qui est chargé d'une base de données digne de son nom, c'est la raison pour laquelle on a corrigé la syntaxe dans SQL:2011. On a inclus des liens de cette manière :

```
SELECT Vendeur, AVG(TOTAL_VENTE)
FROM VENTES
GROUP BY Vendeur
ORDER BY AVG(TOTAL_VENTE) DESC
FETCH FIRST 3 ROWS WITH TIES;
```

À présent, le résultat est complètement déterminé : s'il y a un lien, vous obtiendrez toutes les lignes liées. Comme avant, si vous vous arrêtez au modificateur `WITH TIES`, le résultat sera indéterminé.

Mais SQL:2011 apporte aussi deux autres améliorations à `FETCH` limité.

La première est que les pourcentages sont traités comme un nombre spécifique de lignes. Considérez cet exemple :

```
SELECT Vendeur, AVG(TOTAL_VENTE)
FROM VENTES
GROUP BY Vendeur
ORDER BY AVG(TOTAL_VENTE) DESC
FETCH FIRST 10 PERCENT ROWS ONLY;
```

Un problème de liens peut apparaître lorsque l'on traite des pourcentages comme de simples numéros d'enregistrements, c'est pourquoi la syntaxe `WITH TIES` est ici utilisée. Vous pouvez ou non inclure des liens, tout dépend de ce que vous souhaitez faire dans une situation donnée.

Deuxièmement, supposez que vous ne souhaitiez pas obtenir les trois premiers ou les dix premiers pourcentages, mais le second ou le troisième pourcentage. Peut-être voulez-vous passer directement à un point qui vous semble plus intéressant. SQL:2011 vous permet aussi de traiter ce type de cas. Considérez le code ci-dessous :

```
SELECT Vendeur, AVG(TOTAL_VENTE)
FROM VENTES
GROUP BY Vendeur
ORDER BY AVG(TOTAL_VENTE) DESC
         OFFSET 3 ROWS
FETCH NEXT 3 ROWS ONLY;
```

Le mot clé `OFFSET` indique le nombre de lignes à sauter avant de récupérer. Le mot clé `NEXT` spécifie que les lignes à récupérer sont celles qui se situent immédiatement après l'offset. Puis le nom du vendeur qui a la quatrième, cinquième ou sixième moyenne la plus élevée de ventes est retourné. Comme vous pouvez le constater, sans la syntaxe `WITH TIES`, il y aurait encore un problème d'indétermination. Si le troisième ou quatrième ou cinquième vendeur sont liés, est indéterminé lequel est traité dans le premier lot, lequel dans le second lot.

Il vaut mieux éviter d'utiliser la capacité de limitation de `FETCH`, car elle risque de retourner des résultats qui induisent en erreur.

Regarder par une fenêtre pour créer un jeu de résultats

Les fenêtres et les fonctions de fenêtre ont été d'abord introduites dans SQL:1999. Grâce à une fenêtre, l'utilisateur peut partitionner un ensemble de données, éventuellement classer les lignes par partitions et spécifier une suite de lignes (le cadre de la fenêtre) qui est associée à une ligne donnée.

Le cadre de la fenêtre de la ligne `R` est un sous-ensemble de la partition contenant `R`. Par exemple, le cadre de la fenêtre peut se composer de toutes

les lignes du début à la fin de la partition y compris R, en fonction de la manière dont sont classées les lignes dans la partition.

Une fonction de fenêtre calcule une valeur de la ligne R en fonction des lignes dans le cadre de R.

Par exemple, supposons que vous ayez une table VENTES qui comporte des colonnes CLIENT_ID, NO_FACTURE et TOTAL_VENTE. Le responsable du magasin veut connaître le montant total des achats qu'a fait chaque client sur un nombre spécifique de factures. Vous pouvez obtenir ces informations à l'aide du code suivant :

```
SELECT CLIENT_ID, NO_FACTURE,
SUM (TOTAL_VENTE) OVER
( PARTITION BY CLIENT_ID
ORDER BY NO_FACTURE
ROWS BETWEEN
UNBOUNDED PRECEDING
AND CURRENT ROW )
  FROM VENTES;
```

La clause OVER détermine le nombre de lignes de la requête qui sont partitionnées avant le traitement. Ici c'est la fonction SUM. Une partition est allouée à chaque client. Chaque partition a une liste de numéros de facture, chacune associée à la somme des valeurs de TOTAL_VENTE d'une rangée spécifique de ligne, pour chaque client.

Dans SQL:2011 la fonctionnalité originale de fenêtre a connu d'importantes améliorations, notamment avec l'apparition de nouveaux mots clés.

Partitionner une fenêtre en paquets avec NTILE

La fonction de fenêtre NTILE distribue les lignes d'une partition triée dans un nombre entier positif de groupes. Les groupes sont numérotés à partir de un à *n*. Si le nombre de lignes dans une partition *m* n'est pas divisible par *n*, alors une fois que la fonction NTILE a distribué un nombre à chaque

groupe, le reste *m/n*, appelé *r*, est réattribué au premier groupe *r*, le rendant ainsi un groupe plus grand que les autres.

Supposez que vous souhaitiez classer vos employés par salaire, en les classant dans cinq groupes de salaire du plus haut au plus bas. Vous le ferez avec le code suivant :

```
SELECT PRENOM, NOM, NTILE (5)
          OVER (ORDER BY SALAIRE DESC)
          AS BUCKET
FROM EMPLOYE;
```

Si vous avez 11 employés, chaque groupe comptera deux employés, à l'exception du premier, qui en comportera trois. Le premier groupe sera celui des trois employés les mieux payés, et le cinquième groupe aura les deux employés les moins bien payés.

Naviguer dans une fenêtre

SQL:2011 comporte cinq fonctions de fenêtre qui évaluent une expression dans une ligne `R2` qui se trouve quelque part dans le cadre de la fenêtre de la ligne actuelle `R1`. Ces fonctions sont `LAG`, `LEAD`, `NTH_VALUE`, `FIRST_VALUE` et `LAST_VALUE`.

Ces fonctions vous permettent de retrouver des informations dans des lignes spécifiées qui sont à l'intérieur du cadre de la fenêtre de la ligne actuelle.

Fonction LAG

La fonction `LAG` vous permet de récupérer des informations de la ligne actuelle de la fenêtre que vous examinez ainsi que des informations d'une autre ligne que vous avez spécifiée et qui précède la ligne courante.

Supposez que vous disposiez d'une table qui enregistre les ventes totales de chaque jour de l'année en cours. Vous pourrez ainsi connaître le montant total des ventes par jour et comparer le montant des ventes d'aujourd'hui à celui d'hier par exemple. Vous le ferez en utilisant la fonction `LAG` comme suit :

```
SELECT TOTAL_VENTE AS VENTE_DU_JOUR,
LAG (TOTAL_VETE) OVER
(ORDER BY DATE_VENTE) AS PrevDaySale
FROM TOTAL_JOUR;
```

Pour chaque ligne de TOTAL_JOUR, la requête retournera une ligne listant la ligne du total du jour et celle qui calcule le total de la veille. L'offset par défaut est à 1, c'est la raison pour laquelle le résultat de la veille est retourné et non celui d'un autre jour.

Pour comparer les ventes du jour à celles d'une semaine avant, utilisez le code suivant :

```
SELECT TOTAL_VENTE AS VENTE_DU_JOUR,
LAG (TOTAL_VENTE, 7) OVER
(ORDER BY DATE_VENTE) AS PrevDaySale
    FROM TOTAL_JOUR;
```

Les sept premières lignes dans le cadre de la fenêtre n'auront pas de précédent antérieur à une semaine. La réponse par défaut dans ce cas est de retourner un résultat null pour PrevDaySale. Si vous souhaitez faire apparaître un résultat autre que null, spécifiez que vous voulez voir retourner la valeur 0, par exemple, et non la valeur null comme suit :

```
SELECT TOTAL_VENTE AS VENTE_DU_JOUR,
LAG (TOTAL_VENTE, 7, 0) OVER
(ORDER BY DATE_VENTE) AS PrevDaySale
          FROM TOTAL_JOUR;
```

Par défaut, ici ne sont comptabilisées que les lignes qui sont très grandes, comme c'est le cas de TOTAL_VENTE, qui peut contenir une valeur nulle. Si vous voulez sauter toutes les lignes qui ont une valeur nulle et ne prendre en compte que celles qui ont une valeur réelle, ajoutez le mot clé IGNORE NULLS comme dans cette variante de l'exemple précédent :

```
SELECT TOTAL_VENTE AS VENTE_DU_JOUR,
LAG (TOTAL_VENTE, 7, 0) IGNORE NULLS
```

```
OVER (ORDER BY DATE_VENTE) AS PrevDaySale
FROM TOTAL_JOUR;
```

Fonction LEAD

La fonction LEAD fonctionne exactement comme la fonction LAG, excepté qu'elle n'examine pas la ligne précédente, mais la suivante. En voici un exemple ;

```
SELECT TOTAL_VENTE AS VENTE_DU_JOUR,
LEAD (TOTAL_VENTE, 7, 0) IGNORE NULLS
OVER (ORDER BY DATE_VENTE) AS NextDaySale
FROM TOTAL_JOUR;
```

Fonction NTH_VALUE

La fonction NTH_VALUE est semblable aux fonctions LAG et LEAD, à l'exception près qu'elle n'évalue pas une expression dans une ligne précédente ou suivante de la ligne en cours, mais qu'elle évalue une expression dans une ligne qui est à un offset spécifié de la première ou dernière ligne dans le cadre de la fenêtre.

En voici un exemple :

```
SELECT TOTAL_VENTE AS CHOIX_VENTE,
NTH_VALUE (TOTAL_VENTE, 2)
FROM FIRST
IGNORE NULLS
OVER (ORDER BY DATE_VENTE)
ROWS BETWEEN 10 PRECEDING AND 10 FOLLOWING )
AS PREMIERE_VENTE
FROM TOTAL_JOUR;
```

Dans cet exemple, PREMIERE_VENTE est évaluée ainsi :

» Le cadre de la fenêtre associé à la ligne en cours est formé. Ce qui inclut les dix lignes précédentes et suivantes.

- `TOTAL_VENTE` est évalué dans chaque ligne du cadre de la fenêtre.
- `IGNORE NULLS` est spécifié, toutes les lignes contenant une valeur nulle pour `TOTAL_VENTE` sont sautées.
- On commence après la première valeur qui reste une fois les valeurs nulles pour `TOTAL_VENTE` exclues, on avance de deux lignes à la fois parce que `FROM FIRST` était spécifié.

La valeur `PREMIERE_VENTE` est la valeur de `TOTAL_VENTE` dans une ligne spécifiée.

Si vous ne voulez pas sauter les lignes qui ont une valeur nulle pour `TOTAL_VENTE`, spécifiez `RESPECT NULLS` à la place de `IGNORE NULLS`. La fonction `NTH_VALUE` fonctionne de la même manière, que vous précisiez `FROM LAST` ou `FROM FIRST`, sauf si vous comptez à rebours à partir du dernier enregistrement et non à partir du premier enregistrement. Le nombre de lignes à compter est toujours positif, que vous comptiez à rebours ou en avant.

Fonctions FIRST_VALUE et LAST_VALUE

Les fonctions `FIRST_VALUE` et `LAST_VALUE` sont des cas spéciaux de la fonction `NTH_VALUE` ; FIRST_VALUE est l'équivalent de `NTH_VALUE` où `FROM FIRST` est spécifié et l'offset vaut 0 (zéro). `LAST_VALUE` est l'équivalent de `NTH_VALUE` où `LAST_VALUE` est spécifié et l'offset vaut `0`. Pour les deux fonctions, vous avez le choix de respecter ou d'ignorer les valeurs nulles.

Imbriquer des fonctions de fenêtre

Imbriquer une fonction dans une autre peut être quelquefois la solution dont vous avez besoin. Dans SQL:2011, cette possibilité a été ajoutée et étendue

aux fonctions de fenêtre.

Un investisseur cherche à déterminer si c'est le bon moment d'acheter un stock particulier. Afin de bien arrêter son choix, il décide de comparer le prix du stock actuel aux prix des 100 derniers fournisseurs qui l'ont vendu. Il se demande de combien est moins cher le prix des 100 fournisseurs qui l'ont vendu par rapport au prix actuel. Pour avoir une réponse satisfaisante, il effectue la requête suivante :

```
SELECT DATE_VENTE,
SUM ( CASE WHEN PRIX_VENTE <
VALUE OF (PRIX_VENTE AT CURRENT ROW)
THEN 1 ELSE 0 )
OVER (ORDER BY DATE_VENTE
ROWS BETWEEN 100 PRECEDING AND CURRENT ROW )
FROM STOCK_VENTE;
```

La fenêtre inclut les 100 lignes précédant la ligne en cours, lesquelles correspondent aux 100 ventes immédiatement antérieures au moment actuel. À chaque fois qu'une ligne où la valeur de PRIX_VENTE est inférieure au prix de vente actuel, on ajoute 1 à la somme. Le résultat final est un nombre qui indique le nombre de ventes parmi les 100 précédentes qui ont un prix inférieur au prix actuel.

Évaluer un groupe de lignes

Il peut arriver que les clés de tri que vous avez choisies pour classer une partition peuvent avoir des doubles. Pour évaluer toutes les lignes qui ont la même clé de tri sous forme de groupe, utilisez l'option `GROUPS`. Vous pourrez ainsi compter les groupes de lignes qui ont une clé de tri identique.

En voici un exemple :

```
SELECT CLIENT_ID, DATE_VENTE,
SUM (TOTAL_FACTURE) OVER
( PARTITION BY CLIENT_ID
ORDER BY DATE_VENTE
GROUPS BETWEEN 2 PRECEDING AND 2 FOLLOWING )
```

```
FROM CLIENTS;
```

Le cadre de fenêtre de cet exemple se compose de cinq groupes de lignes : deux groupes avant le groupe contenant la ligne actuelle, le groupe de la ligne actuelle, et deux groupes qui suivent le groupe de la ligne actuelle. Les lignes de chaque groupe ont la même DATE_VENTE et la DATE_VENTE associée à chaque groupe est différente des valeurs de DATE_VENTE des autres groupes.

Chapitre 11
Les opérateurs relationnels

DANS CE CHAPITRE :

- **Combiner des tables de même structure.**
- **Combiner des tables de structures différentes.**
- **Récupérer des données pertinentes depuis plusieurs tables.**

SQL est un langage de requêtes pour les bases de données relationnelles. Dans les précédents chapitres, j'ai présenté des bases de données élémentaires et, dans la plupart des cas, mes exemples ne faisaient référence qu'à une seule table. Il est temps de mettre un peu de *relationnel* dans la base de données ! Après tout, une base est qualifiée de relationnelle quand elle contient plusieurs tables liées entre elles.

Comme une base de données relationnelle contient un ensemble de tables, une requête extrait généralement des informations à partir de plusieurs tables. SQL dispose d'opérateurs qui permettent de combiner des données issues de plusieurs sources au sein d'une seule table résultat. Ce sont les opérateurs `UNION`, `INTERSECT` et `EXCEPT` ainsi que la famille des opérateurs `JOIN`. Chacun d'entre eux combine des données provenant de plusieurs tables d'une manière différente.

UNION

L'opérateur `UNION` est l'implémentation SQL de l'opérateur union de l'algèbre relationnelle. Il vous permet d'extraire des informations de deux ou plusieurs tables de même structure. *De même structure* signifie que :

- Les tables doivent toutes avoir le même nombre de colonnes.
- Les colonnes correspondantes doivent posséder des données de même type et de même longueur.

Lorsque ces conditions sont satisfaites, les tables sont déclarées *compatibles pour une union*. L'union de deux tables retourne toutes les lignes de chaque table et élimine les doublons.

Supposons que vous créiez une base de données pour gérer des statistiques sur le base-ball (comme celle du Chapitre 12). Elle contient deux tables nommées AMERICAINE et NATIONALE qui sont compatibles pour une union. Toutes deux ont trois colonnes, et les colonnes correspondantes ont toujours le même type. De plus, les colonnes correspondantes portent des noms identiques (même si cette condition n'est pas requise pour que ces tables soient déclarées compatibles pour une union).

NATIONALE contient les noms des lanceurs de la Ligue nationale ainsi que le nombre de parties complètes auxquelles ils ont participé. AMERICAINE contient les mêmes informations, mais sur les lanceurs de la Ligue américaine. L'union des deux tables est une table qui contient toutes les lignes de la première table et toutes les lignes de la seconde.

```
SELECT * FROM NATIONALE ;
PRENOM     NOM        PARTIES_COMPLETES
---------- --------- ------------------
Sal        Maglie              11
Don        Newcombe             9
Sandy      Koufax              13
Don        Drysdale            12

SELECT * FROM AMERICAINE ;

PRENOM     NOM       PARTIES_COMPLETES
---------- --------- ------------------
Whitey    Ford               12
Don       Larson             10
```

```
Bob       Turley              8
Allie      Reynolds          14
SELECT * FROM NATIONALE
UNION

SELECT * FROM AMERICAINE ;

PRENOM    NOM        PARTIES_COMPLETES
---------- --------- ------------------
Allie     Reynolds           14
Bob       Turley              8
Don       Drysdale           12
Don       Larson             10
Don       Newcombe            9
Sal       Maglie             11
Sandy     Koufax             13
Whitey    Ford               12
```

La variante `UNION DISTINCT` fonctionne exactement de la même manière. Avec ou sans la précision DISTINCT, l'opérateur `UNION` élimine par défaut toutes les lignes en double du jeu de résultat.

J'ai utilisé un astérisque (`*`) pour adresser plus rapidement toutes les colonnes d'une table. Cette notation peut vous poser des problèmes quand vous utilisez des opérateurs relationnels dans du SQL incorporé ou dans des modules. Que va-t-il se passer si vous ajoutez une ou plusieurs colonnes à une table et pas à l'autre, ou si vous ajoutez des colonnes différentes aux deux tables ? Simplement, elles ne seront plus compatibles pour une union. Votre programme sera donc invalide lors de sa prochaine compilation. Même si deux colonnes identiques sont ajoutées aux deux tables de sorte qu'elles restent compatibles pour une union, votre programme ne sera vraisemblablement pas informé de l'existence de ces nouvelles données. Par conséquent, il vaut toujours mieux dresser la liste explicite des colonnes que vous comptez utiliser plutôt que de se reposer sur le raccourci *. Mais lorsque vous vous contentez de saisir des requêtes SQL depuis la console, l'utilisation de l'astérisque ne pose probablement pas de problème. Si la requête échoue, vous pouvez en effet afficher rapidement la

structure de la table résultat afin de vérifier s'il y a ou non compatibilité pour l'union.

L'opération UNION ALL

Comme je l'ai indiqué plus haut, l'opération `UNION` élimine normalement les doublons, ce qui est la plupart du temps la bonne méthode. Cependant, il peut arriver que vous désiriez conserver ces lignes redondantes. Vous utiliserez alors `UNION ALL`.

Pour reprendre l'exemple précédent, supposons que Bob Turley soit passé en cours de saison de l'équipe New York Yankees de la Ligue américaine à l'équipe Brooklyn Dodgers de la Ligue nationale. Supposons ensuite qu'il ait joué huit parties complètes pour chaque équipe lors de cette saison. L'utilisation `d'UNION` éliminera systématiquement l'une des deux lignes concernant Turley. Même s'il semble qu'il n'ait joué que huit parties complètes lors de cette saison, il a en réalité fait 16 matches en entier. La requête suivante produira le bon résultat :

```
SELECT * FROM NATIONALE
UNION ALL
SELECT * FROM AMERICAINE
```

Vous pouvez parfois former l'union de deux tables, même si elles ne sont normalement pas compatibles pour cela. Si les colonnes que vous désirez avoir dans la table résultat sont présentes, et si elles sont compatibles dans les deux tables, vous avez la possibilité d'effectuer une opération `UNION CORRESPONDING`. Seules les colonnes que vous spécifiez alors sont utilisées pour réaliser l'union, et ce sont aussi les seules qui apparaissent dans la table résultat.

L'opération

Les statisticiens du base-ball conservent des statistiques très différentes sur les lanceurs et sur les autres joueurs. Mais dans les deux cas, ils conservent le nom et le prénom de chaque sportif. Bien que ces tables soient incompatibles, vous pouvez procéder à leur union pour ne récupérer que les

noms, les prénoms, et, disons, les erreurs commises par les uns et par les autres :

```
SELECT *
    FROM JOUEURS
UNION CORRESPONDING
(PRENOM, NOM, ERREURS)
SELECT *
    FROM LANCEURS;
```

La table résultante va afficher les noms, prénoms et nombre d'erreurs commises par tous les joueurs, lanceurs ou non. Comme pour une simple `UNION`, tous les doublons sont éliminés. Dans l'hypothèse où un joueur aurait changé de poste en cours de saison, une partie des statistiques qui le concernent seraient donc perdues. Pour éviter ce problème, vous disposez de la variante `UNION ALL CORRESPONDING`.

Chaque nom de colonne cité dans la liste qui suit le mot clé `CORRESPONDING` doit exister dans les deux tables. Si vous omettez cette liste, toutes les colonnes qui apparaissent dans les deux tables seront utilisées. Mais cette liste implicite peut changer si des colonnes sont ajoutées ultérieurement à l'une ou aux deux tables. C'est pourquoi il vaut toujours mieux dresser une liste explicite des noms des colonnes que vous souhaitez utiliser.

INTERSECT

L'opérateur `UNION` produit une table résultat qui contient *toutes* les lignes qui apparaissent dans *n'importe laquelle* des tables sources. Si vous ne voulez faire apparaître que les lignes qui apparaissent dans toutes les tables sources, vous pouvez utiliser l'opérateur `INTERSECT`, qui est l'implémentation SQL de l'opérateur intersection de l'algèbre relationnelle. Reprenons l'exemple précédent :

```
SELECT * FROM NATIONALE;
PRENOM       NOM       PARTIES_COMPLETES
---------- --------- -----------------
```

```
Sal         Maglie           11
Don         Newcombe          9
Sandy       Koufax           13
Don         Drysdale         12
Bob         Turley            8

SELECT * FROM AMERICAINE;
PRENOM      NOM      PARTIES_COMPLETES
---------- --------- -----------------
Whitey    Ford               12
Don       Larson             10
Bob       Turley              8
Allie     Reynolds           14
```

Avec l'instruction qui suit, seules les lignes apparaissant dans toutes les tables sources figurent dans la table résultat :

```
SELECT *
    FROM NATIONALE
INTERSECT
SELECT *
    FROM AMERICAINE;

PRENOM      NOM      PARTIES_COMPLETES
---------- --------- -----------------
Bob         Turley           8
```

La table résultat nous indique que Bob Turley est le seul lanceur à avoir participé au même nombre de parties complètes dans les deux ligues. Note : comme avec `UNION`, `INTERSECT DISTINCT` retourne le même résultat que l'opérateur `INTERSECT` utilisé seul. Dans cet exemple, seule la ligne mentionnant `Bob Turley` est retournée.

Les mots clés `ALL` et `CORRESPONDING` fonctionnent avec un opérateur `INTERSECT` de la même manière qu'avec `UNION`. Si vous utilisez `ALL`,

les lignes redondantes apparaissent dans la table résultat. Si vous utilisez CORRESPONDING, les tables dont vous calculez l'intersection n'ont pas besoin d'être compatibles pour l'union, même si les colonnes correspondantes doivent être du même type et de la même longueur.

Voici ce que vous obtiendrez avec INTERSECT ALL :

```
SELECT *
    FROM NATIONAL
INTERSECT ALL
SELECT *
    FROM AMERICAINE;

PRENOM    NOM      PARTIES_COMPLETESs
--------- -------- -------------
Bob       Turley    8
Bob       Turley    8
```

Prenons un autre exemple. Une municipalité gère la liste des téléphones portables qui équipent ses élus, ses officiers de police, ses pompiers, ses chefs de service et autres employés municipaux. Une table appelée TELVILLE contient des informations sur tous les appareils en service. Une autre table nommée HS, dont la structure est identique, contient les données sur tous les portables qui, pour une raison ou une autre, ont été déclarés inutilisables. Aucun téléphone ne devrait figurer simultanément dans les deux tables. Vous pouvez utiliser un opérateur INTERSECT pour vérifier que cette règle est bien appliquée :

```
SELECT *
    FROM TELVILLE
INTERSECT CORRESPONDING (TELEPHONE_ID)
SELECT *
    FROM HS ;
```

Si la table résultat contient des lignes, vous savez que vous avez un problème. Car cela signifie qu'un téléphone est à la fois en service et hors service, ce qui est incohérent. Une fois le défaut localisé, vous pouvez

corriger la situation en exécutant une opération DELETE sur une des tables afin de restaurer l'intégrité de la base.

EXCEPT

L'opérateur UNION traite deux tables sources et retourne toutes les lignes qui apparaissent dans les deux tables. L'opérateur INTERSECT retourne toutes les lignes qui apparaissent à la fois dans les deux tables. Par contraste, l'opérateur EXCEPT (ou EXCEPT DISTINCT) retourne toutes les lignes qui apparaissent dans la première table, mais pas dans la seconde.

Reprenons l'exemple des téléphones portables de la base de données municipale. Certains appareils avaient été déclarés hors service et retournés chez le fabricant pour réparation. Ils sont maintenant rentrés et remis en service. La table TELVILLE a été mise à jour pour y réintégrer les téléphones réparés. Mais, pour une raison quelconque, ces appareils n'ont pas été supprimés de la table HS, alors qu'ils auraient dû l'être. Vous pouvez afficher les numéros TELEPHONE_ID de la table HS tout en éliminant les références des téléphones remis en service à l'aide de l'opérateur EXCEPT :

```
SELECT *
    FROM HS
EXCEPT CORRESPONDING (TELEPHONE_ID)
SELECT *
    FROM TELVILLE;
```

Cette requête retourne toutes les lignes de la table HS pour lesquelles TELEPHONE_ID est aussi présent dans la table TELVILLE.

Les opérateurs JOIN

Les opérations UNION, INTERSECT et EXCEPT sont très utiles pour manipuler plusieurs tables qui sont compatibles pour une union. Mais vous allez souvent devoir extraire des données de tables qui ont bien peu de

choses en commun. Les *jointures* (`JOIN`) sont des opérateurs relationnels très puissants qui permettent de combiner des données extraites de plusieurs sources disparates afin de former une table résultat. SQL dispose de plusieurs types de jointures. Toute la difficulté consiste à choisir celui qui correspond le mieux au résultat que vous souhaitez obtenir. Nous allons tenter de nous y retrouver dans les sections qui suivent.

Jointure élémentaire

Toute requête qui porte sur plusieurs tables est un type de jointure. Les tables sources sont jointes en ce sens que la table résultat contient des informations extraites de toutes les tables sources. Le plus simple des `JOIN` est un `SELECT` sans clause `WHERE` appliqué à deux tables. Chaque ligne de la première table est jointe à chaque ligne de la seconde table. Le résultat est le produit cartésien des deux tables sources. Le nombre de lignes que contient la table résultat est égal au nombre de lignes de la première table source multiplié par le nombre de lignes de la seconde table source.

Imaginons par exemple que vous soyez le DRH d'une société et qu'une partie de votre travail consiste à gérer des informations sur les membres du personnel. La plupart des données qui concernent un employé, comme l'adresse de son domicile et son numéro de téléphone, ne sont pas particulièrement confidentielles. Mais d'autres données, comme son salaire, ne doivent être consultées que par les personnes autorisées. Pour protéger ces informations, vous allez les stocker dans une table distincte dont l'accès est protégé par un mot de passe. Considérez ces deux tables :

```
EMPLOYES    INDEMNITES
--------  ----------
EMP_ID     EMPLOYE
PRENOM     SALAIRE
NOM        PRIMES
VILLE
TELEPHONE
```

Remplissons les tables avec ces exemples :

```
EMP_ID    PRENOM    NOM     VILLE    TELEPHONE
------    ------    ---     ------    ---------
1         Whitey    Ford      Orange   555-1001
2         Don       Larson    Newark   555-3221
3         Sal       Maglie    Nutley   555-6905
4         Bob       Turley    Passaic  555-8908
EMPLOYE    SALAIRE   PRIMES
-------    --------  ------
1           33000     10000
2           18000      2000
3           24000      5000
4           22000      7000
```

Créez maintenant une table résultat à l'aide de la requête suivante :

```
SELECT *
    FROM EMPLOYES, INDEMNITES ;
```

Ce qui produit :

```
EMP_ID PRENOM    NOM     VILLE    TELEPHONE
EMPLOYE    SALAIRE   PRIMES
------ ------   ------  ------   ---------    ---
----    -------    ------
1    Whitey      Ford    Orange    555-1001    1
33000    10000
1    Whitey      Ford    Orange    555-1001    2
18000     2000
1    Whitey      Ford    Orange    555-1001    3
24000     5000
1    Whitey      Ford    Orange    555-1001    4
22000     7000
2    Don        Larson   Newark    555-3221    1
33000    10000
```

```
2    Don         Larson   Newark    555-3221    2
18000     2000
2    Don         Larson   Newark    555-3221    3
24000     5000
2    Don         Larson   Newark    555-3221    4
22000     7000
3    Sal         Maglie   Nutley    555-6905    1
33000    10000
3    Sal         Maglie   Nutley    555-6905    2
18000     2000
3    Sal         Maglie   Nutley    555-6905    3
24000     5000
3    Sal         Maglie   Nutley    555-6905    4
22000     7000
4    Bob         Turley   Passaic   555-8908    1
33000    10000
4    Bob         Turley   Passaic   555-8908    2
18000     2000
4    Bob         Turley   Passaic   555-8908    3
24000     5000
4    Bob         Turley   Passaic   555-8908    4
22000     7000
```

La table résultat, qui est le produit cartésien d'EMPLOYES et d'INDEMNITES, contient beaucoup de lignes redondantes. En outre, sa signification n'est pas claire. Elle combine chaque ligne d'EMPLOYES avec chaque ligne d'INDEMNITES. Les seules lignes significatives sont celles qui ont un numéro EMP_ID extrait d'EMPLOYES correspondant à un numéro EMPLOYE provenant d'INDEMNITES. Dans ce cas seulement, le nom et l'adresse d'un employé sont associés à son salaire.

Lorsque vous essayez d'extraire des informations d'une base de données qui contient plusieurs tables, le produit cartésien retourné par une jointure élémentaire n'est presque jamais ce que vous attendez. Cependant, en appliquant des contraintes aux JOIN avec des clauses WHERE, vous

pouvez éliminer toutes les lignes indésirables. Le `JOIN` le plus courant qui utilise la clause `WHERE` pour filtrer est une jointure équivalente.

Equijointure

Une *jointure équivalente (*ou *équijointure*) est une jointure élémentaire dont la clause `WHERE` contient une condition qui spécifie qu'une valeur d'une colonne de la première table doit être égale à la valeur de la colonne correspondante de la seconde table. En appliquant une équijointure aux tables de l'exemple précédent, vous obtiendrez un résultat bien plus pertinent :

```
SELECT *
    FROM EMPLOYES, INDEMNITES
    WHERE EMPLOYES.EMP_ID = INDEMNITES.EMPLOYE ;
```

Ce qui produit :

```
EMP_ID  PRENOM    NOM     VILLE   TELEPHONE
EMPLOYE  SALAIRE PRIMES
------  ------    ----    ------   ---------  ---
----  ------- ------
1       Whitey    Ford    Orange   555-1001
1     33000    10000
2       Don       Larson  Newark   55-3221
2     18000    2000
3       Sal       Maglie  Nutley   555-6905
3     24000    5000
4       Bob       Turley  Passaic  555-8908
4     22000    7000
```

Dans cette table de résultat, les salaires et les primes qui figurent à droite sont associés aux employés dont les noms figurent à gauche. Mais la table contient encore quelques données redondantes, car la colonne `EMP_ID` est répétée dans la colonne `EMPLOYES`. Vous pouvez résoudre ce problème en reformulant légèrement la requête :

```
SELECT
EMPLOYES.*,INDEMNITES.SALAIRE,INDEMNITES.PRIMES
    FROM EMPLOYES, INDEMNITES
    WHERE EMPLOYES.EMP_ID = INDEMNITES.EMPLOYE ;
```

Ce qui produit :

```
EMP_ID    PRENOM     NOM    VILLE    TELEPHONE
SALAIRE   PRIMES
------    ------   ------  -------   ---------
-------   ------
1         Whitey   Ford     Orange   555-1001
33000     10000
2         Don     Larson    Newark   555-3221
18000      2000
3         Sal     Maglie    Nutley   555-6905
24000      5000
4         Bob     Turley    Passaic  555-8908
22000      7000
```

Cette table vous indique ce que vous voulez savoir sans vous encombrer de données superflues. Cependant, la requête est quelque peu fastidieuse à écrire, car pour ôter toute ambiguïté vous avez préfixé chaque colonne par le nom de la table à laquelle elle appartient.

Vous pouvez simplifier l'écriture de la requête en utilisant des *alias* (ou *noms de corrélation*). Un *alias* est l'abréviation du nom d'une table. Si vous utilisez des alias pour écrire la requête précédente, elle ressemblera à ceci :

```
SELECT E.*, I.SALAIRE, I.PRIMES
    FROM EMPLOYES E, INDEMNITES I
    WHERE E.EMP_ID = I.EMPLOYE ;
```

Dans cet exemple, `E` est l'alias `d'EMPLOYES` et `I` celui `d'INDEMNITES`. La portée de l'alias est limitée à celle de l'instruction où il est déclaré. Une fois que vous avez déclaré un alias (dans la clause

FROM), vous ne devez plus utiliser que cet alias dans la requête. Vous ne pouvez plus indiquer à la fois l'alias et le vrai nom de la table.

L'utilisation simultanée du nom initial de la table et de l'alias peut induire en erreur. Considérez la requête suivante :

```
SELECT T1.C, T2.C
    FROM T1 T2, T2 T1
    WHERE T1.C > T2.C ;
```

Dans cet exemple, l'alias de T1 est T2 et l'alias pour T2 est T1. Bien entendu, définir de telles corrélations n'est pas recommandable, mais rien ne l'interdit. Si vous utilisez en même temps les alias et les noms des tables dans la requête, vous ne serez plus en mesure de distinguer les tables que vous manipulez.

L'exemple précédent équivaut au SELECT suivant :

```
SELECT T2.C, T1.C
    FROM T1 , T2
    WHERE T2.C > T1.C ;
```

SQL vous permet d'effectuer la jointure de plus de deux tables. Le nombre maximal de tables que vous pouvez joindre varie d'une implémentation à une autre (mais il est en général plus que suffisant pour couvrir tous vos besoins). La syntaxe utilisée se présente de la manière suivante :

```
SELECT E.*, C.SALAIRE, C.PRIMES, Y.TOTAL_VENTES
    FROM EMPLOYES E, INDEMNITES C, VENTES Y
    WHERE E.EMP_ID = C.EMPLOYE
      AND C.EMPLOYE = Y.NOEMP ;
```

Cette instruction effectue une équijointure sur trois tables pour produire une table résultat qui contient le nom, le montant des ventes et le salaire de chaque commercial. Le directeur des ventes peut ainsi rapidement juger de l'adéquation entre le salaire et les performances d'un commercial.

Pour améliorer les performances et la fiabilité, il est conseillé de stocker les ventes effectuées par les commerciaux dans une table distincte de la table EMPLOYES. En effet, les données de la table EMPLOYES sont

relativement statiques. Le nom, le numéro de téléphone et l'adresse d'un commercial ne changent pas souvent. Par contre, ses résultats sur l'année sont par essence variables. Comme la table VENTES contient moins de colonnes qu'EMPLOYES, vous pourrez la mettre à jour plus rapidement. En outre, cela vous épargnera de modifier les informations contenues dans la table EMPLOYES.

Jointure croisée

`CROSS JOIN` est le mot clé pour désigner une jointure élémentaire dépourvue de clause `WHERE`. Par conséquent :

```
SELECT *
    FROM EMPLOYES, INDEMNITES ;
```

peut aussi s'écrire :

```
SELECT *
    FROM EMPLOYES CROSS JOIN INDEMNITES ;
```

Le résultat est le produit cartésien (également appelé produit vectoriel) des deux tables sources. `CROSS JOIN` produit rarement le résultat que vous escomptiez, mais c'est un traitement qui est souvent utilisé pour effectuer un regroupement préalable de données avant de les manipuler pour construire le résultat final.

Jointure naturelle

Une *jointure naturelle* est un cas particulier d'équijointure. Dans la clause `WHERE` d'une équijointure, une colonne d'une table source est comparée à une colonne d'une seconde table pour déterminer si leurs valeurs sont égales. Ces deux colonnes doivent être de même type et de même longueur, mais peuvent porter des noms différents. En fait, dans une jointure naturelle, toutes les colonnes d'une table qui ont les mêmes nom, type et longueur que les colonnes correspondantes dans la seconde table font l'objet d'une comparaison.

Supposez que la table INDEMNITES de l'exemple précédent contienne les colonnes `EMP_ID`, `SALAIRE` et `PRIMES` à la place de `EMPLOYE`, `SALAIRE` et `PRIMES`. Dans ce cas, vous pouvez effectuer une jointure naturelle entre les tables INDEMNITES et EMPLOYES. La syntaxe de ce `JOIN` serait la suivante :

```
SELECT E.*, I.SALAIRE, I.PRIMES
    FROM EMPLOYES E, INDEMNITES I
    WHERE E.EMP_ID =I.EMP_ID ;
```

Cette requête est une jointure naturelle. Vous pouvez aussi utiliser la syntaxe suivante :

```
SELECT E.*, I.SALAIRE, I.PRIMES
    FROM EMPLOYES E NATURAL JOIN INDEMNITES I ;
```

Cette requête effectue la jointure des lignes où E.EMP_ID = I.EMP_ID, où I.SALAIRE=C.SALAIRE, et où I.PRIMES =C.PRIMES. La table résultat contiendra seulement les lignes dont toutes les colonnes auront des correspondances. Dans cet exemple, le résultat de ces requêtes est le même car la table EPLOYE ne contient ni colonne Salaire ni colonne Prime.

Jointure conditionnelle

Une *jointure conditionnelle* est une équijointure, à ceci près que la condition de test n'est pas l'égalité (bien que ce puisse être le cas), mais un prédicat. Si la condition est satisfaite par une ligne, celle-ci est intégrée à la table résultat. La syntaxe employée est légèrement différente de ce que vous avez pu voir jusqu'à présent. La condition est contenue dans une clause `ON` et non plus dans une clause `WHERE`.

Supposons qu'un statisticien du base-ball désire identifier les lanceurs de la Ligue nationale qui ont participé au même nombre de parties complètes qu'un ou plusieurs des lanceurs de la Ligue américaine. Cette question peut être résolue avec une jointure équivalente, mais aussi avec une jointure conditionnelle :

```
SELECT *
```

```
FROM NATIONALE JOIN AMERICAINE
ON NATIONALE.PARTIES_COMPLETES =
AMERICAINE.PARTIES_COM-
PLETES ;
```

Jointure par nom de colonne

Une *jointure par nom de colonne* est une jointure naturelle très souple d'utilisation. Lors d'une jointure naturelle, toutes les colonnes qui portent le même nom dans les tables sources sont comparées entre elles pour tester leur égalité. Avec une jointure par nom de colonne, il est possible de spécifier les colonnes homonymes qui doivent être comparées et celles qui ne doivent pas l'être. Vous pouvez choisir toutes les colonnes, faisant ainsi de cette jointure par nom de colonne une jointure naturelle. Vous pouvez également ne spécifier que quelques colonnes homonymes, ce qui vous permettra de contrôler finement les lignes qui apparaîtront dans votre table résultat.

Supposons que vous soyez un fabricant d'échiquiers et que vous disposiez d'une table d'inventaire pour gérer votre stock de pièces blanches, et d'une autre table concernant quant à elle votre stock de pièces noires. Ces tables contiennent les données suivantes :

```
BLANC                          NOIR
-----                           -----
Pièce    Quant    Bois         Pièce    Quant
Bois
-----    -----    -----        -----    -----    --
---
Roi       502    Chêne         Roi       502
Ebène
Reine     398    Chêne         Reine     397
Ebène
Pion     1020    Chêne         Pion     1020
Ebène
Fou       985    Chêne         Fou       985
```

```
Ebène
Cavalier  950    Chêne       Cavalier  950
Ebène
Tour      431    Chêne       Tour      453
Ebène
```

Pour chaque type de pièce, le nombre de pièces blanches devrait être égal au nombre de pièces noires. Si ce n'est pas le cas, des échiquiers sont incomplets. Les pièces ont pu être perdues ou volées. Dans tous les cas, des mesures de contrôle s'imposent.

Une jointure naturelle teste l'égalité de toutes les colonnes homonymes. Dans notre exemple, la table résultat ne contiendra rien, car aucune ligne de la colonne `BOIS` de la table BLANC ne correspond à une ligne de la colonne `BOIS` de la table NOIR. Cette table résultat ne vous permet pas de savoir si des pièces sont manquantes. Effectuez plutôt une jointure par nom de colonne en excluant la colonne `BOIS`. La requête peut prendre la forme suivante :

```
SELECT *
    FROM BLANC JOIN NOIR
    USING (PIECE, QUANT) ;
```

La table résultat ne contient que les lignes pour lesquelles le nombre de pièces blanches est égal au nombre de pièces noires.

```
Pièce     Quant  Bois   Pièce     Quant  Bois
-----     -----  -----  -----     -----  -----
Roi        502   Chêne  Roi        502   Ebène
Pion      1020   Chêne  Pion      1020   Ebène
Fou        985   Chêne  Fou        985   Ebène
Cavalier   950   Chêne  Cavalier   950   Ebène
```

La personne qui gère les stocks pourra déduire de cette liste qu'il y a un problème avec les reines et les tours.

Jointure interne

Vous en êtes probablement au stade où vous pensez que les jointures sont une affaire plutôt ésotérique, et qu'il faut un haut degré de discernement spirituel pour s'en servir correctement. Vous avez peut-être aussi entendu parler de la mystérieuse jointure interne, ce qui vous a plongé dans une profonde réflexion sur l'essence même des opérations relationnelles et leurs rapports éventuels avec d'anciennes pratiques chamaniques...

En fait, cette jointure n'a rien de mystérieux. Toutes les jointures présentées dans ce chapitre *sont* des jointures internes. J'aurais pu qualifier la jointure par nom de colonne de l'exemple précédent de *jointure interne* en utilisant la syntaxe suivante :

```
SELECT *
    FROM BLANC INNER JOIN NOIR
    USING (PIECE, QUANT) ;
```

Le résultat obtenu aurait été exactement le même.

Tout simplement, la jointure interne est ainsi appelée pour la distinguer de la jointure externe. Une jointure interne supprime toutes les lignes dans la table résultat qui ne correspondent pas à des lignes équivalentes dans les deux tables sources. La jointure externe conserve ces lignes. C'est là toute la différence ! Il n'y a donc rien de métaphysique dans cette affaire.

Jointure externe

Lorsque vous effectuez la jointure de deux tables, la première (à gauche) peut contenir des lignes sans équivalent dans la seconde table (à droite). De la même manière, la table de droite peut contenir des lignes sans équivalent dans la table de gauche. Si vous effectuez une jointure interne sur ces tables, toutes les lignes sans correspondance seront exclues du résultat. Par contre, une *jointure externe* les conservera. Il existe trois types de jointures externes : la jointure externe gauche, la jointure externe droite et la jointure externe complète.

Jointure externe gauche

Dans une requête qui contient une jointure, la table de gauche est celle qui précède le mot clé `JOIN`, et la table de droite celle qui le suit (mais vous aviez trouvé !). *Une jointure externe gauche* conserve les lignes de la table de gauche qui n'ont pas d'équivalent, mais elle supprime celles de la table de droite qui sont dans le même cas.

Pour mieux comprendre les jointures externes, prenons l'exemple d'une base de données contenant des informations sur les employés, les départements et les sites d'une entreprise. Les Tableaux 11.1, 11.2 et 11.3 contiennent quelques exemples de données.

TABLEAU 11.1 La table SITES.

SITE_ID	VILLE
1	Boston
3	Tampa
5	Chicago

Supposons maintenant que vous souhaitiez visualiser toutes les informations relatives aux employés, y compris leur département et leur site. Vous pouvez obtenir le résultat voulu avec une équijointure :

TABLEAU 11.2 La table DEPARTEMENTS.

DEPT_ID	SITE_ID	NOM
21	1	Ventes
24	1	Admin
27	5	Réparation
29	5	Stock

TABLEAU 11.3 La table EMPLOYES.

EMP_ID	DEPT_ID	NOM
61	24	Kirk
63	27	McCoy

```
SELECT *
    FROM SITES S, DEPARTEMENTS D, EMPLOYES E
    WHERE S.SITE_ID = D.SITE_ID
      AND D.DEPT_ID = E.DEPT_ID ;
```

Cette instruction produit le résultat suivant :

```
1 Boston 24 1 Admin 61 24 Kirk
5 Chicago 27 5 Reparation 63 27 McCoy
```

La table résultat contient toutes les données relatives à tous les employés, dont le site et le département où ils travaillent. L'équijointure fonctionne, car tous les employés travaillent sur un site dans un certain département.

Supposons maintenant que vous souhaitiez obtenir des informations sur les sites. Le problème est différent, car un site peut très bien ne pas avoir de département. Pour obtenir le résultat escompté, vous devez utiliser une jointure externe telle que :

```
SELECT *
    FROM SITES S LEFT OUTER JOIN DEPARTEMENTS D
      ON (S.SITE_ID = D.SITE_ID)
    LEFT OUTER JOIN EMPLOYES E
    ON (D.DEPT_ID = E.DEPT_ID);
```

La jointure extrait des données des trois tables. Tout d'abord, la table SITES est jointe à la table DEPARTEMENTS. Puis la table résultat est à son tour jointe à la table EMPLOYES. Les lignes de la table qui se trouve à gauche de l'opérateur `LEFT OUTER JOIN` et qui n'ont pas d'équivalent dans la table placée à droite de l'opérateur sont incluses dans le résultat. Par conséquent, lors de cette première jointure, tous les sites sont inclus, y compris ceux qui ne sont pas associés à des départements. Lors de la

seconde jointure, tous les départements sont inclus, même si aucun employé n'y est attaché. Le résultat est le suivant :

```
1 Boston 24 1 Admin 61 24 Kirk
5 Chicago 27 5 Reparation 63 27 McCoy
3 Tampa NULL NULL NULL NULL NULL NULL
5 Chicago 29 5 Stock NULL NULL NULL
1 Boston 21 1 Ventes NULL NULL NULL
```

Les deux premières lignes sont identiques à celles de l'exemple précédent. Les colonnes qui correspondent au département et à l'employé de la troisième ligne (3 Tampa) contiennent des valeurs nulles, car aucun département n'est défini à Tampa et aucun employé n'y travaille. Les quatrième et cinquième lignes (5 Chicago et 1 Boston) contiennent des informations sur les départements Ventes et Stock, mais la colonne « Employés » y prend une valeur nulle, car aucune personne n'est rattachée à ces deux départements. La jointure externe vous fournit tout ce que la jointure équivalente vous donnait, plus :

- Tous les sites de la société, qu'ils possèdent ou non un département.
- Tous les départements de la société, qu'ils disposent ou non d'employés.

Les lignes retournées dans l'exemple précédent n'apparaissent pas forcément dans l'ordre que vous souhaitez. Cet ordre peut varier d'une implémentation à une autre. Pour vous assurer que les lignes retournées sont classées comme vous le désirez, ajoutez une clause `ORDER BY` dans votre instruction `SELECT`, comme ceci :

```
SELECT *
    FROM SITES S LEFT OUTER JOIN DEPARTEMENTS D
      ON (S.SITE_ID = D.SITE_ID)
LEFT OUTER JOIN EMPLOYES E
  ON (D.DEPT_ID = E.DEPT_ID)
ORDER BY S.SITE_ID, D.DEPT_ID, E.EMP_ID;
```

Une jointure externe gauche peut également s'écrire `LEFT JOIN`, car il n'existe pas de jointure interne gauche.

Jointure externe droite

Il n'y a plus qu'à intervertir les équipes... La *jointure externe droite* préserve dans la table de droite les lignes sans équivalent dans la table de gauche, et inversement supprime de la table de gauche les lignes sans équivalent dans la table de droite. Vous pouvez l'utiliser sur les mêmes tables et obtenir les mêmes résultats en inversant simplement l'ordre dans lequel vous indiquez les tables lors de la jointure.

```
SELECT *
    FROM EMPLOYES E RIGHT OUTER JOIN DEPARTEMENTS
D
      ON (D.DEPT_ID = E.DEPT_ID)
    RIGHT OUTER JOIN SITES S
      ON (S.SITE_ID = D.SITE_ID) ;
```

Dans cette formule, la première jointure produit une table qui contient tous les départements, qu'ils aient ou non un employé. La seconde produit une table qui contient tous les sites, qu'ils disposent ou non d'un département.

Une jointure externe droite peut également s'abréger en `RIGHT JOIN`, car il n'existe pas de jointure interne droite.

Jointure externe complète

La *jointure externe complète* est une combinaison de jointure externe gauche et de jointure externe droite. Elle retient les lignes sans correspondance à la fois dans les tables de gauche et de droite. Reprenons l'exemple précédent. Notre société peut avoir :

- Des sites sans départements.
- Des départements sans site.
- Des départements sans employés.

» Des employés sans département.

Pour afficher tous les sites, départements et employés, que leurs lignes disposent ou non de lignes équivalentes dans les autres tables, utilisez une jointure externe complète sous la forme suivante :

```
SELECT *
    FROM SITES S FULL JOIN DEPARTEMENTS D
      ON (S.SITE_ID = D.SITE_ID)
    FULL JOIN EMPLOYES E
      ON (D.DEPT_ID = E.DEPT_ID) ;
```

Une jointure externe complète peut également s'abréger en `FULL JOIN`, car il n'existe pas de jointure interne complète.

Jointure d'union

Contrairement aux autres types de jointure, *la jointure d'union* n'essaie pas de comparer une ligne de la table source de gauche aux lignes de la table source de droite. Elle crée une nouvelle table virtuelle qui contient le résultat de l'union des colonnes des deux tables sources.

Dans la table virtuelle résultat, les colonnes extraites de la table source de gauche contiennent toutes les lignes qui se trouvaient dans cette table. Dans ces lignes, les colonnes extraites de la table de droite prennent des valeurs nulles. De même, les colonnes extraites de la table source de droite contiennent toutes les lignes qui se trouvaient dans cette table, et, dans chacune de ces lignes, les colonnes extraites de la table de gauche prennent la valeur nulle. Ainsi la table résultat d'une jointure union reprend toutes les colonnes des deux tables sources, et la quantité de lignes qu'elle contient est égale à la somme du nombre des lignes des deux tables sources.

Le résultat d'une jointure d'union n'est pas immédiatement utilisable dans la plupart des cas, la table produite contenant généralement de nombreuses valeurs nulles. Cependant, vous pouvez utiliser cette table avec l'expression `COALESCE` (voyez le Chapitre 9) pour récupérer des informations intéressantes. Prenons un exemple.

Supposons que vous travailliez pour une société qui conçoit et construit des fusées expérimentales. Vous menez de front plusieurs projets. Vous collaborez avec des ingénieurs dont les compétences sont multiples. En qualité de responsable, vous voulez savoir qui sont ces ingénieurs, de quelles compétences ils disposent et sur quels projets ils travaillent. Actuellement, ces informations sont réparties entre les tables EMPLOYES, PROJETS et COMPETENCES.

La table EMPLOYES contient des données sur les personnels et `EMP_ID` est sa clé primaire. La table PROJETS contient une ligne pour chaque projet auquel un employé a participé. `PROJETS. EMP_ID` est une clé étrangère qui fait référence à la table EMPLOYES. La table COMPETENCES décrit les compétences de chaque employé. `COMPETENCES.EMP_ID` est une clé étrangère qui référence la table EMPLOYES.

La table EMPLOYES contient exactement une ligne pour chaque employé. Les tables PROJETS et COMPETENCES ont zéro ou plusieurs lignes.

Les Tableaux 11.4, 11.5 et 11.6 présentent des exemples de données pour ces trois tables.

TABLEAU 11.4 La table EMPLOYES.

EMP_ID	NOM
1	Ferguson
2	Frost
3	Toyon

TABLEAU 11.5 La table PROJETS.

NOM_PROJET	EMP_ID
X-63	Structure 1
X-64	Structure 1
X-63	Guidage 2

X-64	Guidage 2
X-63	Télémétrie 3
X-64	Télémétrie 3

TABLEAU 11.6 La table COMPETENCES.

COMPETENCE	EMP_ID
Conception mécanique	1
Aérodynamique	1
Conception analogique	2
Gyroscope	2
Conception numérique	3
Conception R/F	3

À la lecture de ces tableaux, vous apprenez que Ferguson a travaillé à la construction des structures des X-63 et X-64, et qu'il est un expert en conception mécanique et en aérodynamique.

Vous souhaitez maintenant visualiser toutes les informations relatives à tous les employés. Vous décidez d'appliquer une équijointure aux tables EMPLOYES, PROJETS et COMPETENCES.

```
SELECT *
    FROM EMPLOYES E, PROJETS P, COMPETENCES C
    WHERE E.EMP_ID = P.EMP_ID
      AND E.EMP_ID = C.EMP_ID ;
```

Vous pouvez exprimer cette même opération en utilisant une jointure interne de la manière suivante :

```
SELECT *
```

```
    FROM EMPLOYES E INNER JOIN PROJETS P
  ON (E.EMP_ID = P.EMP_ID)
INNER JOIN COMPETENCES C
  ON (E.EMP_ID = C.EMP_ID) ;
```

Les deux formulations donnent le même résultat, celui qui est représenté dans le Tableau 11.7.

TABLEAU 11.7 Résultat de la jointure interne.

E.EMP_ID	E.NOM	P.EMP_ID	P.NOM_PROJET	C.EMP_ID	C.COMPETENCE
1	Ferguson	1	X-63 Structure	1	Conception mécanique
1	Ferguson	1	X-63 Structure	1	Aérodynamique
1	Ferguson	1	X-64 Structure	1	Conception mécanique
1	Ferguson	1	X-64 Structure	1	Aérodynamique
2	Frost	2	X-63 Guidage	2	Conception analogique
2	Frost	2	X-63 Guidage	2	Gyroscope
2	Frost	2	X-64 Guidage	2	Conception analogique
2	Frost	2	X-64 Guidage	2	Gyroscope

3	Toyon	3	X-63 Télémétrie	3	Conception numérique
3	Toyon	3	X-63 Télémétrie	3	Conception R/F
3	Toyon	3	X-64 Télémétrie	3	Conception numérique
3	Toyon	3	X-64 Télémétrie	3	Conception R/F

Ces données ne vous apprennent pas grand-chose. Les numéros identifiant les employés apparaissent quatre fois chacun et les lignes relatives aux compétences d'un employé sont dupliquées. La jointure interne n'est donc pas particulièrement adaptée pour répondre à ce type de question. Vous pouvez obtenir un meilleur résultat en utilisant une jointure d'union combinée à quelques instructions `SELECT` soigneusement choisies. Commençons par une jointure d'union de base :

```
SELECT *
FROM EMPLOYES E UNION JOIN PROJETS P
UNION JOIN COMPETENCES C ;
```

Remarquez l'absence de clause `ON`. La jointure d'union ne filtre pas les données. C'est pourquoi cette clause n'est pas nécessaire. L'instruction produit le résultat représenté dans le Tableau 11.8.

TABLEAU 11.8 Résultat de UNION JOIN.

E.EMP_ID	E.NOM	P.EMP_ID	P.NOM_PROJET	C.EMP_ID	C.COMPETENCE
1	Ferguson	NULL	NULL	NULL	NULL
NULL	NULL	1	X-63 Structure	NULL	NULL

NULL	NULL	1	X-64 Structure	NULL	NULL
NULL	NULL	NULL	NULL	1	Conception mécanique
NULL	NULL	NULL	NULL	1	Aérodynamique
2	Frost	NULL	NULL	NULL	NULL
NULL	NULL	2	X-63 Guidage	NULL	NULL
NULL	NULL	2	X-64 Guidage	NULL	NULL
NULL	NULL	NULL	NULL	2	Conception analogique
NULL	NULL	NULL	NULL	2	Gyroscope
3	Toyon	NULL	NULL	NULL	NULL
NULL	NULL	3	X-63 Télémétrie	NULL	NULL
NULL	NULL	3	X-64 Télémétrie	NULL	NULL
NULL	NULL	NULL	NULL	3	Conception numérique
NULL	NULL	NULL	NULL	3	Conception R/F

Chaque table a été étendue vers la droite ou vers la gauche avec des valeurs nulles, et les lignes ainsi étendues ont été réunies. Notez que l'ordre des lignes est arbitraire et dépend de l'implémentation. Vous pouvez maintenant organiser les données sous une forme plus exploitable.

Remarquez que la table comporte trois colonnes ID dont une seule ne contient pas de valeur nulle dans chaque ligne. Vous pouvez améliorer l'affichage en fusionnant ces colonnes ID. Comme je l'ai dit au Chapitre 9, l'expression `COALESCE` sélectionne la première valeur non nulle dans une

liste. Ici, nous allons lui demander de prendre la valeur du seul numéro d'identification non nul dans la liste de colonnes qui lui est fournie :

```
SELECT COALESCE (E.EMP_ID, P.EMP_ID, C.EMP_ID) AS
ID,
      E.NOM, P.NOM_PROJET, C.COMPETENCES
    FROM EMPLOYES E UNION JOIN PROJETS P
      UNION JOIN COMPETENCES C
    ORDER BY ID ;
```

La clause `FROM` est la même que dans les exemples précédents, mais vous fusionnez les trois colonnes `EMP_ID` en une seule colonne appelée ID. Vous triez le résultat selon ID. Le Tableau 11.9 vous montre le résultat obtenu.

TABLEAU 11.9 Résultat de la jointure d'union avec l'expression COALESCE.

ID	E.NOM	P.NOM_PROJET	C.COMPETENCE
1	Ferguson	X-63 Structure	NULL
1	Ferguson	X-64 Structure	NULL
1	Ferguson	NULL	Conception mécanique
1	Ferguson	NULL	Aérodynamique
2	Frost	X-63 Guidage	NULL
2	Frost	X-64 Guidage	NULL
2	Frost	NULL	Conception analogique
2	Frost	NULL	Gyroscope
3	Toyon	X-63Télémétrie	NULL
3	Toyon	X-64 Télémétrie	NULL

3	Toyon	NULL	Conception numérique
3	Toyon	NULL	Conception R/F

```
SELECT COALESCE (E.EMP_ID, P.EMP_ID, C.EMP_ID) AS
ID,
      E.NOM, COALESCE (P.TYPE, C.TYPE) AS TYPE,
```

Chacune des lignes de ce résultat contient des données relatives à un projet ou à une compétence, mais pas aux deux à la fois. Pour lire ce résultat, vous devez tout d'abord déterminer de quoi parle chaque ligne (projet ou compétence). Si la colonne `NOM_PROJET` ne contient pas une valeur nulle, la ligne correspond à un projet. Si la colonne COMPETENCES ne contient pas une valeur nulle, la ligne correspond à une compétence.

Vous pouvez clarifier ce résultat en ajoutant une autre expression `COALESCE` à l'instruction `SELECT`, comme suit :

```
  NOM_PROJET, C.COMPETENCES
FROM EMPLOYES E
  UNION JOIN (SELECT "Projet" AS TYPE, P.*
                      FROM PROJETS) P
  UNION JOIN (SELECT "Compétence" AS TYPE, C.*
                      FROM COMPETENCES) C
ORDER BY ID, TYPE ;
```

Dans la jointure d'union, la table PROJETS est remplacée par un `SELECT` imbriqué qui rajoute une colonne appelée `P.TYPE`, dont la valeur constante est « `Projet` », aux colonnes provenant de la table PROJETS. De même, la table COMPETENCES a été remplacée par un `SELECT` imbriqué qui rajoute une colonne appelée `C.TYPE,` dont la valeur constante est « `Compétence` », aux colonnes extraites de la table COMPETENCES. Dans chaque ligne, donc, `P.TYPE` est soit nulle, soit « `Projet` », et `C.TYPE` est soit nulle, soit « `Compétence` ».

Le `SELECT` externe utilise `COALESCE` pour fusionner ces deux colonnes types en une seule appelée `TYPE`. Vous spécifiez ensuite `TYPE` dans la clause `ORDER BY` afin de trier les lignes qui ont le même `ID` pour faire apparaître d'abord les projets, puis les compétences. Le résultat est montré dans le Tableau 11.10.

TABLEAU 11.10 Résultats affinés de la jointure d'union avec des expressions COALESCE.

ID	E.NOM	TYPE	NOM_PROJET	COMPETENCE
1	Ferguson	Projet	X-63 Structure	NULL
1	Ferguson	Projet	X-64 Structure	NULL
1	Ferguson	Compétences	NULL	Conception mécanique
1	Ferguson	Compétences	NULL	Aérodynamique
2	Frost	Projet	X-63 Guidage	NULL
2	Frost	Projet	X-64 Guidage	NULL
2	Frost	Compétences	NULL	Conception analogique
2	Frost	Compétences	NULL	Gyroscope
3	Toyon	Projet	X-63 Télémétrie	NULL
3	Toyon	Project	X-64 Télémétrie	NULL
3	Toyon	Compétences	NULL	Conception numérique
3	Toyon	Compétences	NULL	Conception R/F

La table résultat contient maintenant un exposé très lisible des compétences et des projets sur lesquels ont travaillé les employés enregistrés dans la table EMPLOYES.

Étant donné le grand nombre d'opérations de jointure qu'il est possible d'effectuer, établir des liens entre différentes tables ne devrait pas être un

problème quelle que soit leur structure. À partir du moment où les données existent dans votre base, SQL vous permettra de les retrouver et de les afficher sous une forme significative.

ON et WHERE

La fonction des clauses `ON` et `WHERE` dans les différents types de jointure peut prêter à confusion. Ces différents points devraient vous permettre de clarifier la situation :

- La clause `ON` s'utilise dans des jointures interne, gauche, droite et complète. Les jointures croisées et union ne comportent pas de clause `ON`, car aucune d'entre elles ne filtre les données.
- La clause `ON` dans une jointure interne est l'équivalent logique de la clause `WHERE`. La condition énoncée pourrait être spécifiée avec une clause ON ou une clause `WHERE`.
- La clause `ON` joue dans les jointures externes (gauche, droite et complète) un rôle différent des clauses `WHERE`. `WHERE` ne filtre que les lignes retournées par la clause `FROM`. Les lignes rejetées par le filtre ne sont pas incluses dans le résultat. À l'inverse, la clause `ON` filtre d'abord les lignes issues du produit cartésien, puis les inclut dans le résultat en les étendant éventuellement à l'aide de valeurs nulles.

Chapitre 12

Effectuer des recherches approfondies avec des requêtes imbriquées

DANS CE CHAPITRE :

- **Extraire des données de plusieurs tables avec une seule instruction SQL.**
- **Comparer une valeur d'une table avec un ensemble de valeurs d'une autre table.**
- **Utiliser `SELECT` pour comparer une valeur d'une table avec une valeur d'une autre table.**
- **Comparer une valeur d'une table avec toutes les valeurs correspondantes d'une autre table.**
- **Créer des requêtes qui mettent en corrélation deux lignes correspondantes dans des tables.**
- **Déterminer les lignes à mettre à jour, à supprimer ou à insérer en utilisant une sous-requête.**

L'une des meilleures méthodes pour protéger l'intégrité de vos données est de normaliser votre base de données, ce qui préviendra l'apparition d'anomalie lors des modifications. La *normalisation* consiste à diviser une table en plusieurs autres tables, chacune traitant d'un thème particulier.

Par exemple, vous n'allez pas mélanger des informations sur les produits avec des informations sur les clients au sein d'une même table, même si les clients ont acheté des produits. Si vous normalisez une base de données correctement, les données seront réparties entre des tables multiples. La plupart des requêtes que vous effectuerez alors devront extraire des données

de deux ou plusieurs tables. Pour cela, vous utiliserez une jointure ou l'un des opérateurs relationnels (UNION, INTERSECT ou EXCEPT). Ces opérateurs extraient des informations de plusieurs tables et les combinent en une seule table.

Une autre solution pour extraire des données de deux ou plusieurs tables consiste à utiliser des *requêtes imbriquées*. En SQL, une requête imbriquée est une requête où une instruction externe contient une *sous-requête*. Cette sous-requête peut elle-même incorporer une autre sous-requête de plus bas niveau. Le nombre de requêtes que vous pouvez ainsi imbriquer est en théorie infini, mais il est dans la pratique limité par certaines implémentations.

Les sous-requêtes sont invariablement des instructions SELECT, mais les instructions qui les incluent peuvent aussi être une opération INSERT, UPDATE ou DELETE.

Comme une sous-requête est parfaitement capable de traiter une table différente de celle sur laquelle porte l'instruction qui la contient, les requêtes imbriquées sont un excellent moyen pour extraire des informations provenant de multiples tables.

Supposons par exemple que vous interrogiez la base de données de votre société pour trouver tous les responsables commerciaux âgés de plus de 50 ans. Si vous utilisez les jointures (étudiées au Chapitre 11), votre requête pourra se présenter ainsi :

```
SELECT D.NODEPT, D.NOM, E.NOM, E.AGE
    FROM DEPT D, EMPLOYES E
    WHERE D.RESPONSABLE_ID = E.ID AND E.AGE > 50
;
```

D est l'alias de la table DEPT et E l'alias de la table EMPLOYES. Cette dernière contient une colonne ID qui joue le rôle de clé primaire, tandis que la table DEPT possède une colonne RESPON-SABLE_ID dont la valeur est celle de l'ID de l'employé responsable du département. J'emploie une seule jointure (la liste des tables de la clause FROM) pour coupler les tables, et un WHERE pour filtrer toutes les lignes à l'exception de celles qui correspondent aux critères. Remarquez que la liste des paramètres de

l'instruction `SELECT` mentionne les colonnes `NODEPT` et `NOM` de la table DEPT, et les colonnes `NOM` et AGE de la table EMPLOYES.

Supposons maintenant que vous souhaitiez récupérer le même ensemble de lignes, mais en vous limitant aux colonnes de la table DEPT. Autrement dit, vous voulez obtenir la liste des départements dont le responsable est âgé de plus de 50 ans, mais l'identité et l'âge exact des personnes ne vous intéressent pas. Vous pouvez reformuler la requête en remplaçant la jointure par une sous-requête :

```
SELECT D.NODEPT, D.NOM
    FROM DEPT D
    WHERE EXISTS (SELECT * FROM EMPLOYES E
      WHERE E.ID = D.RESPONSABLEID AND E.AGE >
50) ;
```

Cette requête contient deux nouveaux éléments : le mot clé `EXISTS` et le `SELECT *` dans la clause `WHERE`. Le second `SELECT` est une sous-requête (ou *sous-sélection*), et le mot clé `EXISTS` est l'un des nombreux outils que vous pouvez utiliser dans une sous-requête.

Que fait une sous-requête ?

Les *sous-requêtes* se trouvent généralement dans la clause `WHERE` de l'instruction qui les englobe. Leur fonction consiste à spécifier les critères de recherche de la clause `WHERE`. Différents types de sous-requêtes produiront différents résultats. Certaines renvoient une liste de valeurs que l'instruction externe utilise comme entrée. D'autres retournent une unique valeur que l'instruction englobante évalue à l'aide d'un opérateur de comparaison. Un troisième type de sous-requête produit une valeur `True` ou `False`.

Requêtes imbriquées retournant un ensemble de lignes

Pour illustrer comment une requête imbriquée retourne un ensemble de lignes, supposons que vous travailliez pour un intégrateur de matériel informatique. Votre société, Zetec Corporation, assemble des systèmes à partir de composants achetés, puis vend ces systèmes à des sociétés ou des administrations. Cette activité est décrite dans une base de données relationnelle. Elle contient de nombreuses tables, mais nous ne nous intéresserons qu'à trois d'entre elles : PRODUITS, COMP_UTI-LISES, COMPOSANTS. La table PRODUITS (voir le Tableau 12.1) contient la liste de tous les produits que vous proposez. La table COMPOSANTS (voir le Tableau 12.2) regroupe la liste de tous les composants que vous utilisez. La table COMP_UTILISES (voir Tableau 12.3) précise les composants utilisés dans chaque produit. Ces tables sont définies comme suit :

Un composant peut être utilisé dans plusieurs produits et un produit peut contenir plusieurs composants (relation plusieurs à plusieurs), ce qui peut engendrer des problèmes d'intégrité. Pour contourner cette difficulté, la table de liaison COMP_UTI-LISES permet de relier les tables COMPOSANTS et PRODUITS. Un composant peut apparaître dans plusieurs lignes de COMP_ UTILISES, mais chaque ligne de COMP_UTILISES ne référence qu'un seul composant (relation un à plusieurs). De même, un produit peut apparaître dans plusieurs lignes de COMP_UTI-LISES, mais chaque ligne de COMP_UTILISES ne référence qu'un seul produit (autre relation un à plusieurs). En ajoutant cette table de liaison, une relation plusieurs à plusieurs a été transformée en deux relations un à plusieurs. Cette réduction de complexité de relation est un exemple de normalisation.

Les sous-requêtes introduites par le mot clé IN

Une des formes de sous-requête imbriquée compare une valeur unique à un ensemble de valeurs retournées par `SELECT`. Elle fait appel au prédicat `IN`, selon la syntaxe suivante :

```
SELECT liste_colonnes
    FROM table
    WHERE expression IN (sous-requête) ;
```

TABLEAU 12.1 La table PRODUITS.

Colonne	Type	Contraintes
MODELE	Char (6)	PRIMARY KEY
NOM_PROD	Char (35)	
DESC_PROD	Char (31)	
PRIX	Numeric (9,2)	

TABLEAU 12.2 La table COMPOSANTS.

Colonne	Type	Contraintes
COMP_ID	CHAR (6)	PRIMARY KEY
TYPE_COMP	CHAR (10)	
DESC_COMP	CHAR (31)	

TABLEAU 12.3 La table COMP_UTILISES.

Colonne	Type	Contraintes
MODELE	CHAR (6)	FOREIGN KEY (pour PRODUITS)
COMP_ID	CHAR (6)	FOREIGN KEY (pour COMPOSANTS)

L'évaluation de l'expression qui Figure dans la clause `WHERE` produit une valeur. Si cette dernière appartient à la liste retournée par la sous-requête, la clause `WHERE` retourne une valeur Vrai, et les colonnes de la ligne traitée sont ajoutées au résultat. La sous-requête peut référencer la même table que la requête externe ou une autre table.

La base de données de Zetec illustre l'utilisation de ce type de requête. Supposez que l'industrie informatique soit frappée par une pénurie de moniteurs. Si votre stock est épuisé, vous ne serez plus en mesure de livrer les produits qui comportent des écrans. Vous voulez donc savoir quels sont ces produits. Saisissez la requête suivante :

```
SELECT MODELE
    FROM COMP_UTILISES
    WHERE COMP_ID IN
      (SELECT COMP_ID
      FROM COMPOSANTS
      WHERE TYPE_COMP = 'Moniteur') ;
```

SQL exécute la requête la plus imbriquée en premier, c'est-à-dire celle qui retourne la valeur de `COMP_ID` de chaque ligne de la table COMPOSANTS où le TYPE_COMP est `'Moniteur'`. Le résultat est une liste des numéros d'identification de tous les moniteurs. La requête externe compare ensuite la valeur du `COMP_ID` de chaque ligne de la table `COMP_UTILISES` aux valeurs de cette liste. Si la comparaison est validée, la valeur de la colonne MODELE de cette ligne est ajoutée à la table résultat produite par le `SELECT` externe. Le résultat final est une liste de tous les produits qui comportent un moniteur. L'exemple suivant vous montre ce qui se passe lors de l'exécution de cette requête :

```
MODELE
------
CX3000
CX3010
CX3020
MB3030
MX3020
MX3030
```

Vous connaissez maintenant les produits pour lesquels vous serez bientôt en rupture de stock.

Si vous utilisez cette forme d'imbrication, la sous-requête doit spécifier une seule colonne, et le type de donnée de cette colonne doit correspondre au type de l'argument précédant le mot clé `IN`.

Les sous-requêtes introduites par le mot

clé NOT IN

Vous pouvez tout aussi bien introduire une sous-requête avec le mot clé `NOT IN`. En utilisant la requête décrite dans la section précédente, la direction de Zetec a identifié les produits qui ne peuvent plus être commercialisés faute de moniteurs. C'est intéressant, mais pas très efficace sur le plan commercial. Elle veut donc maintenant savoir quels sont les produits *qui ne comportent pas* de moniteur afin de renforcer leur vente. Une requête imbriquée comportant une sous-requête introduite par le mot clé `NOT IN` fournira le résultat escompté :

```
SELECT MODELE
    FROM COMP_UTILISES
    WHERE MODELE NOT IN
      (SELECT COMP_ID
         FROM COMPOSANTS
         WHERE TYPE_COMP = 'Moniteur')) ;
```

Cette requête produit le résultat suivant :

```
MODELE
------
PX3040
PB3050
PX3040
PB3050
```

Notez que la table résultat contient des entrées dupliquées. Cette duplication est due au fait qu'un produit contenant plusieurs composants autres que des moniteurs possède une ligne dans la table COMP_UTILISE pour chaque composant utilisé. La requête crée une entrée dans la table résultat pour chacune de ces lignes.

Dans cet exemple, le nombre de lignes ne crée pas de problème, car la table résultat est de taille réduite. Mais cette table pourrait contenir des centaines, voire des milliers de lignes. Vous devriez alors éliminer les doublons. Pour cela, ajoutez dans la requête le mot clé `DISTINCT`. Seules

les lignes distinctes (différentes) de toutes celles qui ont déjà été récupérées seront ajoutées à la table résultat :

```
SELECT DISTINCT MODELE
    FROM COMP_UTILISES
    WHERE MODELE NOT IN
     (SELECT COMP_ID
      FROM COMPOSANTS
      WHERE TYPE_COMP = 'Moniteur')) ;
```

Comme prévu, le résultat est le suivant :

```
MODELE
------
PX3040
PB3050
```

Requêtes imbriquées retournant une seule valeur

L'introduction d'une sous-requête par l'un des six opérateurs de comparaison (`=`, `<>`, `<`, `<=`, `>`, `>=`) se révèle souvent utile. L'expression qui précède l'opérateur est évaluée pour obtenir une valeur unique. Il en va de même pour la sous-requête qui suit l'opérateur. Il existe cependant une exception à cela : l'emploi d'un *opérateur de comparaison quantifié*, qui est un opérateur de comparaison suivi d'un quantificateur (`ANY`, `SOME` ou `ALL`).

Reprenons l'exemple de la base de données de Zetec. Cette base de données possède une table CLIENTS chargée de mémoriser des informations sur les sociétés qui achètent des produits Zetec. Elle comporte également une table CONTACTS contenant des informations confidentielles sur les personnes qui travaillent dans les entreprises clientes de Zetec. Les Tableaux 12.4 et 12.5 représentent la structure de ces tables.

TABLEAU 12.4 La table CLIENTS.

Type	Colonne	Contraintes
CLIENT_ID	INTEGER	PRIMARY KEY
SOCIETE	CHAR (40)	
ADRESSE_CLIENT	CHAR (30)	
VILLE_CLIENT	CHAR (20)	
ETAT_CLIENT	CHAR (2)	
CP_CLIENT	CHAR (10)	
TEL_CLIENT	CHAR (12)	
NIV_MOD	INTEGER	

TABLEAU 12.5 La table CONTACTS.

Type	Colonne	Contraintes
CLIENT_ID	INTEGER	FOREIGN KEY
CONT_NOM	CHAR (10)	
CONT_PRENOM	CHAR (16)	
CONT_TELEPHONE	CHAR (12)	
CONT_INFO	CHAR (50)	

Supposez que vous recherchiez votre contact chez Olympic Sales, mais vous ne vous rappelez plus du `CLIENT_ID` de cette société. Vous pouvez utiliser la requête suivante :

```
SELECT *
    FROM CONTACTS
  WHERE CLIENT_ID =
      (SELECT CLIENT_ID
         FROM CLIENTS
```

```
                    WHERE SOCIETE = ‘Olympic Sales’) ;
```

Le résultat obtenu est le suivant :

```
CLIENT_ID CONT_PRENOM   CONT_NOM    CONT_TELEPHON
E CONT_INFO
--------- -----------   --------    -------------
-  ------------
118       Jerry         Attwater    505-876-3456
Va jouer un rôle

majeur dans la

mise en oeuvre du

Web sans fil.
```

Si vous utilisez une sous-requête dans une comparaison ‘=’, la liste spécifiée dans la sous-requête `SELECT` ne doit mentionner qu’une seule colonne (par exemple `CLIENT_ID`). Lorsqu’elle est exécutée, elle doit retourner une seule ligne de manière à produire une valeur unique.

Dans cet exemple, je suppose que la table CLIENTS ne contient qu’une ligne dont la valeur de la colonne `SOCIETE` est ‘`Olympic Sale`s’. Si l’instruction `CREATE TABLE` spécifie la contrainte `UNIQUE` pour SOCIETE, la sous-requête de l’exemple précédent retournera systématiquement une seule valeur (ou aucune). Toutefois, ce type de sous-requête est largement utilisé avec des colonnes qui ne sont pas spécifiées comme étant `UNIQUE`.

Vous ne pouvez alors que vous fier à votre connaissance de la base de données pour présumer que la colonne n’est pas dupliquée.

Si la colonne `SOCIETE` de plusieurs clients prend la valeur ‘`Olympic Sale`s’ (peut-être dans différentes régions), la sous-requête générera une erreur.

Par ailleurs, s'il n'y a aucun client pour la société indiquée, la sous-requête est traitée comme si elle générait une valeur nulle et le résultat de la comparaison devient inconnu. Dans ce cas, la clause `WHERE` ne retourne aucune ligne (car elle ne donne que les lignes pour lesquelles la condition est satisfaite, et élimine celles pour lesquelles la condition est fausse ou indéterminée). Cette situation peut survenir si une personne orthographie mal le nom de la SOCIETE en tapant par exemple `'Olumpic Sales'`.

Même si l'opérateur d'égalité `(=)` est le plus utilisé, vous pouvez utiliser les cinq autres opérateurs de la même manière. Pour chaque ligne de la table spécifiée dans l'instruction englobant la clause `FROM`, la valeur retournée par la sous-requête est comparée à celle de l'expression figurant dans la clause `WHERE` de l'instruction de niveau supérieur. Si la comparaison renvoie une valeur True, la ligne est ajoutée à la table résultat.

Vous pouvez vous assurer que la sous-requête retournera une seule valeur en utilisant une fonction d'agrégation. En effet, les fonctions d'agrégation retournent toujours une seule valeur (reportez-vous au Chapitre 3 pour plus de détails). Bien entendu, cette manière de procéder n'est utile que si vous voulez le résultat produit par une telle fonction.

Supposons maintenant que vous soyez commercial chez Zetec. Vous voulez faire un bon chiffre d'affaires (pour régler quelques factures en retard) et vous concentrez vos ventes sur le produit le plus cher de Zetec. Vous pouvez identifier ce produit avec la requête suivante :

```
SELECT MODELE, NOM_PROD, PRIX
    FROM PRODUITS
  WHERE PRIX =
      (SELECT MAX(PRIX)
          FROM PRODUITS) ;
```

Dans cet exemple de requête imbriquée, la sous-requête et l'instruction qui l'englobe travaillent sur la même table. La sous-requête retourne une unique valeur : le prix maximal trouvé dans la table PRODUITS. La requête externe retourne toutes les lignes de la table PRODUITS où Figure ce prix.

L'exemple suivant est une sous-requête qui utilise un opérateur de comparaison autre que `=` :

```
SELECT MODELE, NOM_PROD, PRIX
    FROM PRODUITS
  WHERE PRIX <
      (SELECT AVG(PRIX)
          FROM PRODUITS) ;
```

La sous-requête retourne une valeur : le prix moyen de la table PRODUITS. La requête externe retourne toutes les lignes de la table PRODUITS dans lesquelles le prix est inférieur ou égal à cette moyenne.

Dans le standard initial de SQL, une comparaison ne pouvait avoir qu'une seule sous-requête, et celle-ci devait se placer à droite de la comparaison. SQL:1999 permet d'utiliser des sous-requêtes à droite comme à gauche de l'opérateur de comparaison.

Les quantificateurs ALL, SOME et ANY

Une autre manière de s'assurer qu'une sous-requête retourne une seule valeur consiste à l'introduire avec un opérateur de comparaison quantifié. Le quantificateur universel `ALL` et les quantificateurs existentiels `SOME` et `ANY` associés à un opérateur de comparaison traitent la liste retournée par une sous-requête et la réduisent à une seule valeur.

Vous allez maintenant voir comment ces quantificateurs affectent une comparaison en examinant la base de données des parties complètes de base-ball présentée au Chapitre 11.

Le contenu des deux tables est retourné par les deux requêtes suivantes :

```
SELECT * FROM NATIONALE
PRENOM      NOM              PARTIES_COMPLETES
------      --------         -----------------
Sal          Maglie           11
Don          Newcombe          9
Sandy        Koufax           13
Don         Drysdale          12
```

```
Bob          Turley            8
SELECT * FROM AMERICAINE
PRENOM       NOM              PARTIES_COMPLETES
------       --------         -----------------
Whitey       Ford              12
Don          Larson            10
Bob          Turley             8
Allie        Reynolds          14
```

En théorie, les lanceurs qui ont participé au plus grand nombre de parties complètes sont les membres de la Ligue américaine. Pour le vérifier, il suffit de construire une requête qui retourne tous les lanceurs de cette ligue qui ont participé à plus de parties complètes que les lanceurs de la Ligue nationale. Cette requête peut se formuler comme suit :

```
SELECT *
    FROM AMERICAINE
    WHERE PARTIES_COMPLETES > ALL
         (SELECT PARTIES_COMPLETES FROM
NATIONALE) ;
```

Le résultat est le suivant :

```
PRENOM     NOM     PARTIES_COMPLETES
------ -------- -----------------
Allie     Reynolds  14
```

La sous-requête (`SELECT PARTIES_COMPLETES FROM NATIONALE`) retourne les valeurs de la colonne `PARTIES_COMPLETES` de tous les lanceurs de la Ligue nationale. Le quantificateur `> ALL` indique qu'il ne faut retourner que les valeurs de `PARTIES_COMPLETES` de la table AMERICAINE qui sont supérieures à toutes les valeurs renvoyées par la sous-requête. La condition peut donc se traduire par « plus grande que la valeur la plus élevée retournée par la sous-requête ». Dans notre cas, la valeur la plus élevée retournée par la sous-requête est `13 (Sandy Koudax)`. La seule ligne dans la table

AMERICAINE dont la valeur de la colonne `PARTIES_COMPLETES` est supérieure à `13` est celle d'Allie Reynolds (il a participé à 14 matches entiers).

Et si votre théorie se révélait fausse ? Dans ce cas, la requête suivante vous signalera qu'aucun des lanceurs de la Ligue américaine n'a participé à plus de parties complètes qu'un lanceur de la Ligue nationale (autrement dit qu'il n'y a aucune ligne à retourner) :

```
SELECT *
    FROM AMERICAINE
      WHERE PARTIES_COMPLETES > ALL
      (SELECT PARTIES_COMPLETES FROM NATIONALE) ;
```

Requêtes imbriquées servant de test d'existence

Une requête retourne des données extraites de toutes les lignes de la table qui satisfont aux conditions posées. Il arrive que de nombreuses lignes soient retournées. Parfois, il y en a simplement une seule. Mais il est aussi possible qu'aucune ligne ne satisfasse les conditions, et donc qu'aucune ligne ne soit renvoyée. Vous pouvez utiliser les prédicats `EXISTS` et `NOT EXISTS` pour introduire une sous-requête. Ils vous indiqueront si la table identifiée dans la clause `FROM` satisfait ou non les conditions énoncées dans la clause `WHERE`.

Les sous-requêtes introduites par `EXISTS` ou `NOT EXISTS` sont fondamentalement différentes des autres sous-requêtes présentées dans ce chapitre. Dans les exemples précédents, SQL exécute en premier la sous-requête, puis utilise le résultat de cette opération à l'instruction qui l'englobe. D'un autre côté, `EXISTS` et `NOT EXISTS` sont des exemples de sous-requêtes corrélées.

Une *sous-requête corrélée* recherche tout d'abord la table et la ligne spécifiée par l'instruction englobante, puis exécute la sous-requête sur la ligne de la table qui correspond à la ligne courante de la table de l'instruction de plus niveau.

La sous-requête retourne soit une ou plusieurs lignes, soit aucune. Si elle renvoie au moins une ligne, le prédicat `EXISTS` est validé, et l'instruction englobante est exécutée. Dans les mêmes conditions, le prédicat `NOT EXISTS` ne sera pas validé et donc l'instruction ne sera pas exécutée. Lorsqu'une ligne de la table de l'instruction englobante a été traitée, l'opération se répète pour la ligne suivante. Et ainsi de suite jusqu'à ce que toutes les lignes de la table de l'instruction englobante aient été passées en revue.

EXISTS

Vous êtes un vendeur de Zetec Corporation, et vous devez téléphoner à tous les contacts des clients de Zetec en Californie. Essayez la requête suivante :

```
SELECT *
    FROM CONTACTS
    WHERE EXISTS
  (SELECT *
      FROM CLIENTS
      WHERE CLIENT_ETAT = 'CA'
         AND CONTACTS.CLIENT_ID =
CLIENTS.CLIENT_ID) ;
```

Remarquez la présence de `CLIENT_ID` qui référence une colonne de la requête externe et la compare à une autre colonne (`CLIENTS.CLIENT_ID`) de la requête interne. Pour chaque ligne de la table CONTACTS traitée par la requête externe, vous évaluez la requête interne en utilisant sa valeur `CLIENT_ID` dans la clause `WHERE` de la requête interne.

Voici comment :

1. La colonne `CLIENT_ID` relie les tables CONTACTS et CLIENTS.
2. SQL se place sur le premier enregistrement de la table CONTACT. Il trouve la ligne de la table CLIENTS qui possède

le même CLIENT_ID, puis teste le champ CLIENT_ETAT de cette ligne.

3. Si CLIENTS. CLIENT_ETAT = 'CA', la ligne courante de CONTACTS est ajoutée à la table résultat.
4. L'enregistrement suivant est traité de la même manière, et ainsi de suite jusqu'à ce que la table CONTACTS ait été entièrement parcourue.
5. Comme la requête spécifie SELECT * FROM CONTACTS, tous les champs de la table sont affichés, y compris le nom et le numéro de téléphone d'un contact.

NOT EXISTS

Dans l'exemple précédent, le vendeur de Zetec recherchait les noms et les numéros de téléphone des contacts de tous les clients résidant en Californie. Imaginez qu'un second vendeur soit responsable de tous les États-Unis à l'exception de la Californie. Il peut récupérer la liste de ses contacts en utilisant NOT EXISTS dans une requête semblable à la précédente :

```
SELECT *
    FROM CONTACTS
    WHERE NOT EXISTS
      (SELECT *
         FROM CLIENTS
         WHERE CLIENT_ETAT = 'CA'
                  AND CONTACTS.CLIENT_ID =
CLIENT.CLI-
            ENT_ID) ;
```

Chaque ligne de CONTACTS pour laquelle la sous-requête ne retourne rien est ajoutée à la table résultat.

Autres sous-requêtes corrélées

Comme je l'ai déjà expliqué plus haut, les sous-requêtes introduites par `IN` ou par un opérateur de comparaison n'ont pas besoin d'être des requêtes corrélées. Mais elles peuvent l'être.

Sous-requêtes corrélées introduites par IN

Dans la section intitulée « Les sous-requêtes introduites par le mot clé `IN` », j'ai expliqué comment des sous-requêtes non corrélées peuvent être utilisées avec le prédicat `IN`. Pour comprendre maintenant comment une sous-requête corrélée est capable d'utiliser le prédicat IN, posons-nous la même question qu'avec le prédicat `EXISTS` : « Quels sont les noms et numéros de téléphone des contacts de Zetec en Californie ? » Vous pouvez répondre à cette question avec une sous-requête corrélée `IN` :

```
SELECT *
  FROM CONTACTS
  WHERE 'CA' IN
(SELECT CLIENT_ETAT
       FROM CLIENTS
       WHERE CONTACTS.CLIENT_ID =
CLIENTS.CLIENT_ID) ;
```

L'instruction est évaluée pour chaque enregistrement de la table CONTACTS. Si, pour cet enregistrement, les numéros CLIENT_ID dans CONTACTS et dans CLIENTS sont identiques, la valeur de `CLIENTS.CLIENT_ETAT` est comparée à `'CA'`. Le résultat de la sous-requête est une liste qui contient au plus un élément. Si cet élément est `'CA'`, la clause `WHERE` de l'instruction de niveau supérieur est validée et la ligne est ajoutée dans la table résultat.

Sous-requêtes introduites par des opérateurs de comparaison

Une sous-requête corrélée peut être également introduite par l'un des six opérateurs de comparaison, comme l'illustre l'exemple suivant.

Zetec verse des primes à ses vendeurs en fonction du volume des ventes qu'ils réalisent par mois. Plus ce volume est important, plus le pourcentage de la prime est important. Ce pourcentage est conservé dans une table appelée PRIMES :

MONTANT_MIN	MONTANT_MAX	PCT_PRIME
0.00	24999.99	0.
25000.00	49999.99	0.1
50000.00	99999.99	0.2
100000.00	249999.99	0.3
250000.00	499999.99	0.4
500000.00	749999.99	0.5
750000.00	999999.99	0.6

Si les ventes mensuelles d'un vendeur sont comprises entre 100 000 et 249 999,99 unités, le pourcentage de la prime s'élève à 0,3 % des ventes.

Les ventes sont enregistrées dans une table appelée TRANSMASTER :

TRANSMASTER

Colonne	Type	Contraintes
TRANS_I	INTEGER	PRIMARY KEY
CLIENT_ID	INTEGER	FOREIGN KEY
EMP_ID	INTEGER	FOREIGN KEY
TRANS_DATE	DATE	
MONTANT_NET	NUMERIC	
PORT	NUMERIC	
TAXES	NUMERIC	
TOTALFACTURE	NUMERIC	

La prime est basée sur la somme des champs `MONTANT_NET` de toutes les transactions effectuées par un vendeur durant le mois. Vous pouvez calculer

la prime d'un vendeur en utilisant une sous-requête corrélée qui fait appel aux opérateurs de comparaison :

```
SELECT PCT_PRIME
    FROM PRIMES
      WHERE MONTANT_MIN <=
         (SELECT SUM (MONTANT_NET)

               FROM TRANSMASTER
                  WHERE EMP_ID = 133)
      AND MONTANT_MAX >=
      (SELECT SUM (MONTANT_NET)
           FROM TRANSMASTER

                WHERE EMP_ID = 133) ;
```

Cette requête est intéressante, car elle contient deux sous-requêtes reliées par le connecteur logique `AND`. Les sous-requêtes utilisent l'opérateur d'agrégation `SUM` qui retourne une seule valeur : le montant total des ventes réalisées par le vendeur numéro `133`. Cette valeur est ensuite comparée aux colonnes `MONTANT_MIN` et `MONTANT_MAX` de la table PRIMES pour obtenir la prime de ce commercial.

Si vous ne connaissez pas la valeur `EMP_ID`, mais le nom du vendeur, vous obtiendrez le même résultat avec une requête un peu plus complexe :

```
SELECT PCT_PRIME
    FROM PRIMES
      WHERE MONTANT_MIN <=
           (SELECT SUM (MONTANT_NET)
        FROM TRANSMASTER
          WHERE EMP_ID = 133)
AND MONTANT_MAX >=
  (SELECT SUM (MONTANT_NET)
         FROM TRANSMASTER
```

```
                    WHERE EMP_ID = 133) ;
```

Cet exemple utilise des sous-requêtes imbriquées dans des sous-requêtes, qui sont à leur tour imbriquées dans une requête globale pour calculer la prime de l'employé dénommé Coffin. Cette structure ne fonctionnera que si vous êtes absolument certain que la société emploie un et un seul vendeur appelé Coffin. Si l'entreprise emploie plusieurs personnes portant le même nom, vous pouvez ajouter des termes dans la clause `WHERE` de la sous-requête la plus imbriquée jusqu'à ce que vous soyez sûr qu'une seule ligne de la table EMPLOYES est sélectionnée.

Sous-requêtes dans une clause HAVING

Exactement comme dans une clause `WHERE`, vous pouvez utiliser une sous-requête corrélée dans une clause `HAVING`. Comme je l'explique au Chapitre 9, une clause `HAVING` est généralement précédée d'une clause `GROUP BY`. La clause `HAVING` se comporte comme un filtre qui écrème les groupes créés par la clause `GROUP BY`. Ceux qui ne remplissent pas les conditions de la clause `HAVING` ne figurent pas dans le résultat. Dans ce cas, la clause `HAVING` est employée pour chaque groupe créé par la clause `GROUP BY`.

En l'absence de `GROUP BY`, la clause `HAVING` est exécutée pour l'ensemble des lignes retourné par la clause `WHERE`, dont le résultat est alors assimilé à un seul groupe. Si ni `WHERE` ni `GROUP BY` ne sont présentes, la clause `HAVING` est exécutée pour toute la table :

```
SELECT PCT_PRIME
    FROM PRIMES
         WHERE MONTANT_MIN <=
(SELECT SUM (MONTANT_NET)
FROM TRANSMASTER
WHERE EMP_ID =
(SELECT EMP_ID
FROM EMPLOYES
WHERE EMP_NOM = 'Coffin'))
AND MONTANT_MAX >=
```

```
(SELECT SUM (MONTANT_NET)
FROM TRANSMASTER
WHERE EMP_ID =
(SELECT EMP_ID

FROM EMPLOYES
WHERE EMP_NOM = 'Coffin'));
```

Cette requête utilise deux alias pour la même table afin de pouvoir récupérer le numéro d'identifiant de tous les vendeurs dont le volume des ventes est au moins deux fois égal au volume des ventes moyennes de tous les autres vendeurs. Cette requête fonctionne comme suit :

1 La requête externe regroupe les lignes de TRANSMASTER par `EMP_ID`. C'est le travail des clauses `SELECT`, `FROM` et `GROUP BY`.

2. La clause `HAVING` filtre ces groupes et calcule pour chaque groupe le `MAX` de la colonne `MONTANT_NET` des lignes de ce groupe.

3. La requête interne évalue deux fois le `MONTANT_NET` moyen de toutes les lignes de TRANSMASTER dont le `EMP_ID` est différent du `EMP_ID` du groupe courant de la requête externe.

Remarquez que, dans la dernière ligne, vous devez référencer deux valeurs `EMP_ID` différentes, si bien qu'il vous faut utiliser des alias différents de TRANSMASTER dans les clauses `FROM` des requêtes internes et externes.

4. Vous pouvez ensuite utiliser ces alias dans la comparaison de la dernière ligne de la requête pour indiquer que vous référencez à la fois le `EMP_ID` de la ligne courante de la

sous-requête interne (TM2. EMP_ID) et le EMP_ID du groupe courant de la sous-requête externe (TM1. EMP_ID).

Instructions UPDATE, DELETE et INSERT

À l'instar des instructions SELECT, les instructions UPDATE, DELETE et INSERT peuvent également comprendre des clauses WHERE. Ces clauses WHERE sont aussi susceptibles de contenir des sous-requêtes, exactement comme les clauses WHERE des instructions SELECT.

Par exemple, Zetec vient tout juste de réaliser une vente importante avec Olympic Sales et veut rétroactivement concéder à Olympic une ristourne de 10 % sur tous ses achats du mois dernier. Vous pouvez accorder ce crédit avec une instruction UPDATE :

```
UPDATE TRANSMASTER
SET MONTANT_NET = MONTANT_NET * 0.9
WHERE CLIENT_ID =
(SELECT CLIENT_ID
FROM CLIENTS
WHERE COMPAGNIE = 'Olympic Sales') ;
```

Vous pouvez également utiliser une sous-requête corrélée dans une instruction UPDATE. Supposez que la table CLIENTS contienne une colonne MOIS_DERNIER_MAX et que Zetec accorde un crédit de 10 % aux clients dont les achats ont dépassé MOIS_ DERNIER_MAX :

```
UPDATE TRANSMASTER TM
SET MONTANT_NET = MONTANT_NET * 0.9
WHERE MONTANT_NET >
(SELECT MOIS_DERNIER_MAX
FROM CLIENT C
WHERE C.CLIENT_ID = TM.CLIENT_ID) ;
```

Remarquez que cette sous-requête est corrélée. La clause WHERE de la dernière ligne référence à la fois le CLIENT_ID de la ligne CLIENTS de la sous-requête et le CLIENT_ID de la ligne TRANSMASTER courante à mettre à jour.

Une sous-requête dans une instruction UPDATE peut également référencer la table qui est en cours de mise à jour. Supposez que Zetec accorde une ristourne de 10 % aux clients dont les achats ont dépassé 10 000 unités :

```
UPDATE TRANSMASTER TM1
    SET MONTANT_NET = MONTANT_NET * 0.9
WHERE 10000 <
      (SELECT SUM(MONTANT_NET)
FROM TRANSMASTER TM2
WHERE TM1.CLIENT_ID = TM2.CLIENT_ID) ;
```

La sous-requête interne calcule le total (SUM) de la colonne MONTANT_NET pour toutes les lignes de TRANSMASTER relatives à un même client. Concrètement, supposons que le client vérifiant CLIENT_ID = 37 possède quatre lignes dans TRANSMASTER avec les valeurs suivantes pour MONTANT_NET : 3 000, 5 000, 2 000 et 1 000. Le total du montant net pour ce code client est donc de 11 000.

L'ordre dans lequel l'instruction UPDATE traite les lignes est défini par votre implémentation et donc n'est pas en règle générale prévisible. L'ordre peut varier en fonction de la manière dont les lignes sont agencées dans le disque. Supposez que votre implémentation prenne les lignes de TRANSMASTER dans cet ordre : celle dont MONTANT_NET vaut 3 000, puis celle dont MONTANT_NET vaut 5 000, et ainsi de suite. Une fois les trois premières lignes de CLIENT_ID 37 traitées, leur valeur MONTANT_ NET devient 2 700 (90 % de 3 000), 4 500 (90 % de 5 000) et 1 800 (90 % de 2 000). Lorsque vous allez passer à la dernière ligne de TRANSMASTER pour le CLIENT_ID 37 pour laquelle MONTANT_NET vaut 1 000, la SOMME retournée par la sous-requête semblera valoir 10 000, c'est-à-dire la SOMME des nouvelles valeurs de MONTANT_NET des trois premières lignes du CLIENT_ID 37 et de

l'ancienne valeur de MONTANT_NET de la dernière ligne du CLIENT_ID 37. Par conséquent, la dernière ligne du CLIENT_ID 37 n'est apparemment pas mise à jour, car la comparaison avec cette somme n'est pas validée (10 000 n'est pas inférieur à SELECT (SUM (MONTANT_NET)). Mais ce n'est pas ainsi que fonctionne une instruction UPDATE lorsqu'une sous-requête référence une table qui est en cours de mise à jour.

Toutes les évaluations des sous-requêtes d'une instruction UPDATE font référence aux anciennes valeurs de la table à mettre à jour. Dans le précédent UPDATE pour le CLIENT_ID 37, la sous-requête retourne 11 000, qui est la somme originale.

La sous-requête d'une clause WHERE fonctionne exactement de la même manière qu'une instruction SELECT ou une instruction UPDATE. Il en va de même pour DELETE ou INSERT. Pour supprimer toutes les transactions relatives à Olympic, utilisez cette instruction :

```
DELETE TRANSMASTER
WHERE CLIENT_ID =
(SELECT CLIENT_ID
FROM CLIENTS
WHERE COMPAGNIE = 'Olympic Sales') ;
```

Comme dans le cas d'UPDATE, les sous-requêtes DELETE peuvent également être corrélées et référencées dans la table à supprimer. Les règles qui valent dans ce cas sont identiques à celles qui s'appliquent aux sous-requêtes UPDATE. Supposons que vous vouliez supprimer toutes les lignes de TRANSMASTER pour les clients dont le MONTANT_NET est supérieur à 10 000 unités :

```
UPDATE TRANSMASTER TM1
WHERE 10000 < (SELECT SUM(NET_AMOUNT)
FROM TRANSMASTER TM2
WHERE TM1. CLIENT_ID = TM2. CLIENT_ID);
```

Cette requête supprime toutes les lignes de TRANSMASTER pour le `CLIENT_ID 37` ainsi que tous les autres clients dont les achats dépassent 10 000 unités. Toutes les références à TRANSMASTER dans la sous-requête portent sur le contenu de cette table avant toute suppression opérée par l'instruction. Par conséquent, et même si vous êtes en train de supprimer la dernière ligne de TRANSMASTER pour le `CLIENT_ID 37`, la sous-requête sera évaluée sur la base de la table TRANSMASTER initiale et retournera `11 000`.

Lorsque vous mettez à jour, supprimez ou insérez des enregistrements dans une base de données, vous prenez le risque de rendre inconsistantes les données de cette table avec les autres tables de la base. Cette inconsistance, appelée *anomalie de modification*, est présentée au Chapitre 5. Si vous supprimez des enregistrements de TRANSMASTER et qu'une table TRANSDETAIL dépend de TRANSMASTER, vous devez supprimer les enregistrements correspondants dans TRANSDETAIL. Cette opération est appelée *suppression en cascade*, car l'effacement d'un enregistrement parent est répercuté sur les enregistrements enfants qui lui sont associés. Si ces suppressions n'avaient pas lieu, les enregistrements enfants deviendraient orphelins.

Si votre implémentation ne prend pas en charge les suppressions en cascade, vous devrez les faire vous-même. Dans ce cas, supprimez d'abord les enregistrements de la table enfant, puis les enregistrements correspondants de la table parent. Vous n'aurez pas ainsi d'enregistrements orphelins dans la table enfant.

Récupérer des modifications avec le langage de manipulation de données

Dans les précédentes sections, vous avez appris comment une instruction `UPDATE`, `DELETE` ou `INSERT` peut inclure une instruction imbriquée dans une clause `WHERE`. SQL:2011 permet d'imbriquer une commande de manipulation de données (telles que `UPDATE`, `INSERT`, `DELETE` ou `MERGE`) dans une instruction `SELECT`. Cette fonctionnalité est appelée *instruction DML*.

Une autre façon de modifier des données consiste à considérer une table avant toute opération `DELETE`, `INSERT`, ou `UPDATE`. Nommez la table

avant toute modification *vieille table* et la table après modification *nouvelle table*. Pendant les modifications, des tables auxiliaires, appelées *tables delta*, sont créées. Une opération `DELETE` crée une vieille table delta qui contient les lignes à supprimer. Une opération `INSERT` crée une nouvelle table delta qui contient les lignes à insérer. Une opération `UPDATE` crée en même temps une vieille et une nouvelle table delta, la vieille pour les lignes à supprimer et la nouvelle pour les lignes à remplacer.

Avec une DML vous pouvez récupérer les informations des tables delta. Supposez que vous supprimiez tous les produits `PRODUIT_ID` entre `1 000` et `1 399` de la ligne produit et que vous souhaitiez obtenir le compte-rendu exact de ce qui se passe quand tous ces produits sont supprimés. Vous pourriez utiliser les lignes de code ci-dessous :

```
SELECT Oldtable.PRODUIT_ID
    FROM OLD TABLE (DELETE FROM PRODUIT
                    WHERE PRODUIT_ID BETWEEN 1000
AND 1399)
    AS Oldtable ;
```

Dans cet exemple, le mot clé `OLD TABLE` spécifie que le résultat de `SELECT` se trouve dans la vieille table delta. Le résultat est la liste des numéros de produits supprimés.

Vous pouvez aussi récupérer une liste d'une nouvelle table delta avec le mot clé `NEW TABLE` qui affiche les numéros de `PRO-DUIT_ID` des lignes insérées par une opération `INSERT` ou mises à jour par une opération `UPDATE`. Comme une opération `UPDATE` crée aussi bien une vieille qu'une nouvelle table, vous pouvez récupérer le contenu de l'une ou de l'autre ou encore des deux avec une DML.

Chapitre 13
Requêtes récursives

DANS CE CHAPITRE :

- **Comprendre la récursivité.**
- **Définir des requêtes récursives.**
- **Savoir utiliser des requêtes récursives.**

L'une des principales critiques formulées à l'encontre de SQL, jusqu'à sa version SQL-92 incluse, était son incapacité à implémenter des traitements récursifs. De nombreux problèmes sont difficiles à résoudre sans recourir à la *récursivité*. Les extensions ajoutées à SQL:1999 permettent d'effectuer des requêtes récursives, accroissant ainsi la puissance du langage. Si l'implémentation de SQL que vous utilisez dispose de ces extensions, vous serez en mesure de résoudre nombre de nouvelles classes de problèmes. Cependant, comme la récursivité n'est pas une fonctionnalité obligatoire de SQL, de nombreuses implémentations ne l'incluent pas.

Qu'est-ce que la récursivité ?

La notion de *récursivité* est apparue il y a bien longtemps dans des langages de programmation tels que Logo, LISP et C++. Dans ces langages, vous pouvez définir une fonction (ensemble constitué d'une ou plusieurs commandes) qui effectue une certaine opération. Le programme principal invoque la *fonction* en utilisant une commande dite *appel de fonction*. Si, au cours de cette opération, la fonction s'appelle elle-même, vous êtes confronté à la forme la plus élémentaire de récursivité.

Un programme très simple dont une des fonctions utilise la récursivité peut suffire à expliquer les joies et les peines de cette technique. Le programme

suivant, écrit en C++, dessine une spirale à l'écran. Il comprend trois fonctions :

- La fonction `ligne (n)` trace une ligne de *n* unités de long.
- La fonction `tourner_gauche (d)` fait tourner la ligne de *d* degrés dans le sens inverse des aiguilles d'une montre.
- La fonction `spirale (segment)` que l'on peut définir comme suit :

```
void spirale(int segment)
{
ligne(segment)
tourner_gauche(90)
spirale(segment + 1)
}  ;
```

Si nous appelons `spirale (1)` depuis le programme principal, les actions suivantes sont entreprises :

`spirale (1)` trace unelignede une unitéde long vers le haut de l'écran.

`spirale (1)` tourne de 90 degrés.

`spirale (1)` appelle `spirale (2).`

`spirale (2)` trace une ligne de deux unités de long vers le côté gauche de l'écran.

`spirale (2)` tourne de `90` degrés.

`spirale (2)` appelle `spirale (3)`.

... Et ainsi de suite.

Pour finir, le programme génère la spirale représentée sur la Figure 13.1.

Houston, nous avons un problème

Certes, notre situation n'est pas aussi critique que celle d'*Apollo 13* lorsque les astronautes durent rejoindre la Terre alors que leur principale réserve d'oxygène avait explosé dans l'espace. Mais un problème existe. Notre petit programme échappe à notre contrôle. Il s'appelle sans cesse pour tracer des lignes de plus en plus longues. Il continuera ainsi jusqu'à ce qu'il ait épuisé les ressources de l'ordinateur, ce qui générera un message d'erreur. Dans le pire des cas, l'ordinateur se plantera.

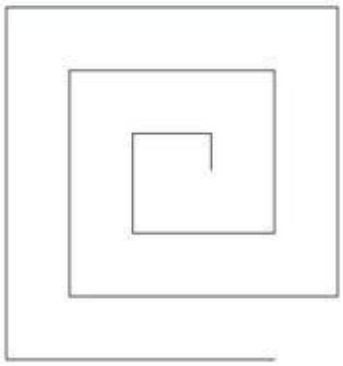


FIGURE 13.1 Résultat de l'appel spirale (1).

Le plantage n'est pas une option

Ce scénario vous explique l'un des dangers de la récursivité. Un programme écrit pour s'appeler invoque une nouvelle instance de lui-même, qui à son tour fait appel à une autre instance de lui-même, et ainsi de suite jusqu'à l'infini. Ce qui n'est généralement pas ce que vous souhaitez.

Pour pallier ce problème, les programmeurs intègrent une condition de sortie dans la fonction récursive, c'est-à-dire une limite à la profondeur des appels récursifs. De cette manière, le programme peut effectuer l'action désirée et se termine sans encombre. Nous allons inclure une telle condition de sortie dans notre programme de dessin de spirale pour sauvegarder les ressources de l'ordinateur et éviter de plonger l'élite des programmeurs dans une dépression noire :

```
void spirale2(int segment)
{
    if (segment <= 10)
```

```
    {
      ligne(segment)
      tourner_gauche(90)
      spirale2(segment + 1)
    }
} ;
```

Lorsque vous faites appel à `spirale2 (1),` la fonction s'exécute, puis s'appelle (récursivement) jusqu'à ce que la valeur du segment dépasse 10. Lorsque `segment =11`, la structure `if (segment<=10)` retourne une valeur Faux et le code à l'intérieur des accolades n'est pas exécuté. Le contrôle revient alors à l'appel précédent de `spirale2`, puis revient à l'appel précédant cet appel précédent, et ainsi de suite jusqu'à ce que le contrôle revienne au premier appel, ce qui met fin au programme.

La Figure 13.2 illustre cette séquence d'appels.

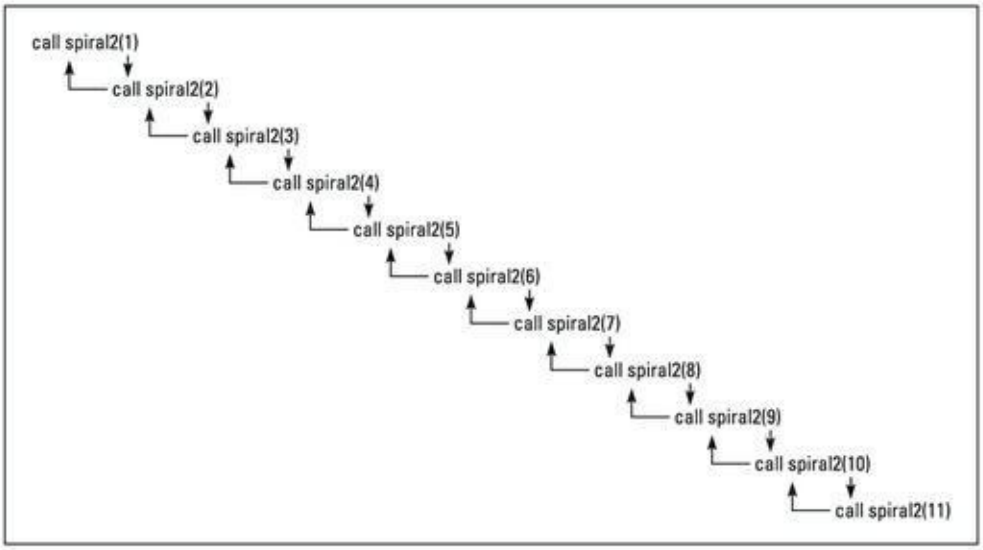


FIGURE 13.2 Une série d'appels descendants puis ascendants pour terminer le programme dans de bonnes conditions.

Chaque fois qu'une fonction s'appelle elle-même, elle vous « enfonce » d'un niveau supplémentaire sous le programme principal qui constitue le point de départ de l'opération. Pour que le programme principal puisse reprendre son cours normal, l'itération la plus « profonde » doit à un moment donné remonter d'un niveau, c'est-à-dire redonner le contrôle à

l'itération qui l'a appelée. Cette itération fera de même, et ainsi de suite jusqu'à revenir au premier appel, puis au programme principal.

La récursivité est un outil puissant pour exécuter du code de manière répétitive lorsque vous ne connaissez pas à l'avance le nombre de boucles à parcourir. Il est idéal pour effectuer des recherches dans des structures arborescentes, des circuits électroniques complexes ou encore des réseaux de distribution possédant de multiples niveaux.

Qu'est-ce qu'une requête récursive ?

Une *requête récursive* est une requête qui dépend fonctionnellement d'elle-même. La forme la plus simple de ce type de dépendance fonctionnelle est la suivante : une requête Q1 inclut un appel à elle-même dans le corps de son expression. La situation devient plus complexe lorsque la requête Q1 dépend de la requête Q2, qui à son tour dépend de la requête Q1. On parle là encore de dépendance fonctionnelle et de récursivité, quel que soit le nombre de requêtes intercalées entre le premier et le second appel à une même requête.

Quand utiliser une requête récursive ?

Les requêtes récursives vous permettent de gagner du temps et d'éviter bien des frustrations lorsque vous essayez de résoudre certains problèmes. Supposons par exemple que vous ayez gagné un passe qui vous permet de voyager gratuitement avec la compagnie aérienne Vannevar Airlines. La première question que vous vous posez est : « Où puis-je aller gratuitement ? » La table VOLS contient tous les vols proposés par Vannevar. Le Tableau 13.1 vous indique leurs numéros, le lieu de départ et le lieu de destination.

TABLEAU 13.1 Les vols proposés par Vannevar Airlines.

N° de vol	Départ	Arrivée

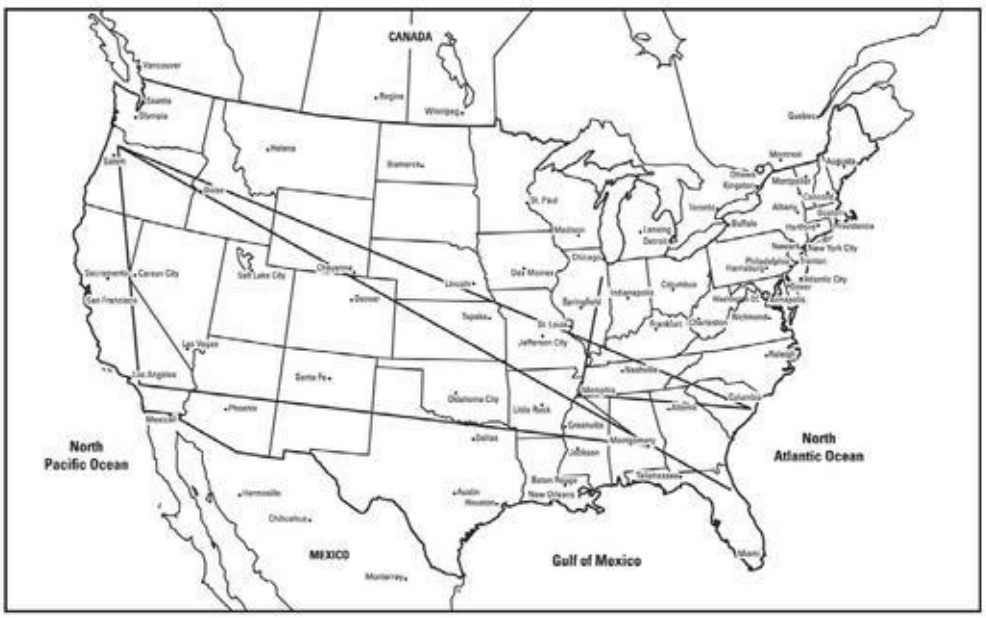
3141	Portland	Orange County
2173	Portland	Charlotte
623	Portland	Daytona Beach
5440	Orange County	Montgomery
221	Charlotte	Memphis
32	Memphis	Champaign
981	Montgomery	Memphis

La Figure 13.3 illustre ces routes sur une carte des États-Unis.

Pour commencer, vous allez créer une table dans la base de données VOLS de la manière suivante :

```
CREATE TABLE VOLS (
NoVol    INTEGER  NOT NULL,

Source    CHARACTER (30),
```



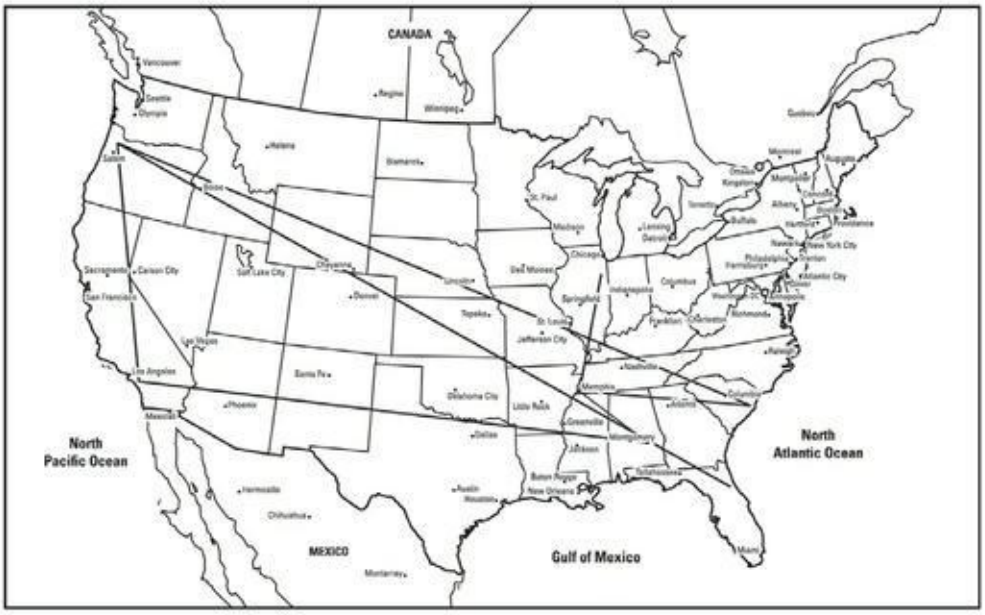


FIGURE 13.3 La carte des vols de Vannevar Airlines.

```
    Destination     CHARACTER (30)
);
```

Vous pourrez ensuite la remplir à l'aide des données listées dans le Tableau 13.1.

Supposons alors que vous partiez de Portland pour rendre visite à un ami qui vit à Montgomery. Vous vous demandez tout naturellement : « Quelles villes puis-je rejoindre depuis Portland par des vols de Vannevar ? » et « Quelles villes puis-je rejoindre depuis Montgomery par des vols de la même compagnie aérienne ? » Vous pouvez relier certaines villes sans escale et d'autres pas. Pour obtenir la liste de toutes les destinations vers lesquelles vous pouvez vous rendre à partir d'une ville donnée (via les vols de Vannevar, évidemment), vous pouvez bien sûr procéder requête après requête, comme ci-dessous. Sauf que vous irez peut-être plus vite avec un papier et un crayon...

La manière forte

Pour obtenir le résultat escompté, et à supposer que vous fassiez preuve de patience, vous pouvez effectuer une série de requêtes en prenant Portland comme point de départ :

```
SELECT Destination FROM VOLS WHERE Source =
‘Portland’ ;
```

Cette première requête va retourner Orange County, Charlotte et Daytona Beach. Votre deuxième requête peut utiliser le premier de ces résultats comme point de départ :

```
SELECT Destination FROM VOLS WHERE Source =
‘Orange Coun-
ty’;
```

Cette deuxième requête renvoie Montgomery. Votre troisième requête va revenir au résultat de la première requête et utiliser le deuxième résultat comme point de départ :

```
SELECT Destination FROM VOLS WHERE Source =
‘Charlotte’ ;
```

La troisième requête donne Memphis. Votre quatrième requête revient au résultat de la première requête et utilise le résultat restant comme point de départ :

```
SELECT Destination FROM VOLS WHERE Source =
‘Daytona
Beach’ ;
```

Hélas ! Cette dernière requête procure un résultat nul, car Vannevar ne propose pas de vol au départ de Daytona Beach. Mais la deuxième requête a retourné une autre ville (Montgomery) qui peut servir de point de départ dans votre cinquième requête :

```
SELECT Destination FROM VOLS WHERE Source =
‘Montgomery’
 ;
```

La réponse est cette fois Memphis. Ce résultat est inutile, car vous savez déjà que Memphis est l’une des villes où vous pouvez aller (dans ce cas,

via Charlotte). Par contre, vous pouvez l'utiliser comme point de départ d'une autre requête :

```
SELECT Destination FROM VOLS WHERE Source =
'Memphis';
```

La requête retourne Champaign, que vous pouvez ajouter à la liste des villes joignables (même s'il faut deux escales pour vous y rendre). Si vous avez envie de découvrir un maximum d'aéroports, vous pouvez tout aussi bien utiliser Champaign comme nouveau point de départ :

```
SELECT Destination FROM VOLS WHERE Source =
'Champaign';
```

Cette requête retourne une valeur nulle, car Vannevar ne propose pas de vol au départ de Champaign. (Sept requêtes plus tard ! Tenez bon !) Vannevar ne propose aucun vol au départ de Daytona Beach, de sorte que si vous vous y rendez, vous ne pourrez plus en partir ! Il ne vous restera plus qu'à vous inscrire à l'université de l'Illinois pour suivre quelques cours sur les bases de données.

En appliquant cette méthode, vous finirez (normalement) par réussir à répondre à la question : « Quelles sont les villes que je peux rejoindre depuis Portland ? » Mais l'enchaînement de requêtes dont le résultat dépend de la précédente se révèle compliqué, pénible et très coûteux en temps de calcul.

Économiser du temps avec une requête récursive

La solution bien plus simple consiste à écrire une seule requête récursive qui effectuera toute la recherche en une seule opération. Voici la syntaxe d'une telle requête :

```
WITH RECURSIVE
    JoignableDepuis (Source, Destination)
      AS (SELECT Source, Destination
                 FROM VOLS
```

```
            UNION
            SELECT sr.Source, dt.Destination
              FROM JoignableDepuis sr, VOLS dt
              WHERE sr.Destination = dt.Source
          )
SELECT * FROM JoignableDepuis
WHERE Source = 'Portland' ;
```

Lors du premier passage récursif, VOLS contient sept lignes et JOIGNABLEDEPUIS aucune. L'opérateur `UNION` prend les sept lignes de VOLS et les copie dans JOIGNABLEDEPUIS. À ce stade, JOIGNABLEDEPUIS contient les données indiquées dans le Tableau 13.2.

Comme je l'ai déjà dit, la récursivité ne fait pas partie de SQL, et par conséquent certaines implémentations ne peuvent pas l'inclure.

TABLEAU 13.2 Le contenu de JOIGNABLEDEPUIS après un passage récursif.

Départ	Arrivée
Portland	Orange County
Portland	Charlotte
Portland	Daytona Beach
Orange County	Montgomery
Charlotte	Memphis
Memphis	Champaign
Montgomery	Memphis

Au deuxième passage récursif, les choses commencent à devenir vraiment intéressantes. La clause `WHERE` (`WHERE sr. Destination = dt. Source`) signifie que vous recherchez seulement les lignes dont le champ `Destination` de la table JOIGNABLEDEPUIS est égal au champ

`Source` de la table VOLS. Pour chacune de ces lignes, vous prenez le champ Source de la table JOIGNABLEDEPUIS et le champ Destination de la table VOLS, puis vous ajoutez ces deux champs à JOIGNABLEDEPUIS pour former une nouvelle ligne. Le Tableau 13.3 vous présente le résultat produit par cette itération.

Ces résultats sont bien plus utiles. JOIGNABLEDEPUIS contient maintenant toutes les villes Destination que vous pouvez rejoindre depuis n'importe quelle ville Source, avec ou sans escale. Le passage récursif suivant permet de traiter tous les voyages à deux escales et ainsi de suite jusqu'à ce que toutes les villes de destination aient été atteintes.

Une fois la recherche récursive terminée, la troisième et dernière instruction SELECT (qui se trouve hors de l'appel récursif) extrait de JOIGNABLEDEPUIS les villes que vous pouvez rejoindre depuis Portland par un vol Vannevar. Dans cet exemple, les six autres villes peuvent être jointes depuis Portland (éventuellement en quelques bonds aériens).

TABLEAU 13.3 Le contenu de JOIGNABLEDEPUIS après deux passages récursifs.

Départ	Arrivée
Portland	Orange County
Portland	Charlotte
Portland	Daytona Beach
Orange County	Montgomery
Charlotte	Memphis
Memphis	Champaign
Montgomery	Memphis
Portland	Montgomery
Portland	Memphis
Orange County	Memphis

Certes, le code de la requête récursive ne paraît pas plus simple que celui des sept requêtes qu'il remplace. Mais il présente deux avantages :

- Une fois son exécution lancée, il effectue toute la recherche sans intervention externe.
- Il peut faire le travail très vite.

Supposez que vous traitiez avec une compagnie aérienne réelle qui dessert bien plus de villes que Vannevar : plus il y aura de destinations, plus vous aurez avantage à utiliser la méthode récursive.

En quoi cette requête est-elle récursive ? Nous avons défini JOIGNABLEDEPUIS en faisant référence à elle-même. La partie récursive de la définition se trouve dans la deuxième instruction `SELECT`, celle qui est juste après l'appel à `UNION`. JOIGNABLEDEPUIS est une table temporaire qui se remplit progressivement de données tandis que se succèdent les passages récursifs.

La recherche se poursuit jusqu'à ce que toutes les destinations possibles aient été ajoutées à JOIGNABLEDEPUIS. Les doublons sont tous éliminés, car l'opérateur `UNION` n'intègre pas les lignes redondantes dans la table résultat. Une fois la recherche récursive achevée, JOIGNABLEDEPUIS contient toutes les destinations que vous pouvez rejoindre depuis n'importe quelle ville de départ. La troisième et dernière instruction `SELECT` fournit enfin la liste des villes accessibles depuis Portland. Bon voyage !

À quoi d'autre peut me servir une requête récursive ?

Tout problème que vous pouvez représenter sous la forme d'une structure arborescente peut éventuellement être résolu à l'aide d'une requête récursive. Une application industrielle des plus classiques est le processus de conversion d'une matière première en produit fini. Supposez que votre entreprise fabrique un nouveau modèle de voiture à moteur hybride essence

et électrique. Un tel véhicule est composé de nombreuses parties (moteur, batterie, etc.) qui sont à leur tour composées de nombreuses pièces plus petites (électrodes, etc.), qui sont à leur tour composées de nombreux éléments encore plus petits.

Il est relativement difficile de gérer correctement ces parties de parties dans une base de données relationnelle qui ne sait pas utiliser la récursivité. Celle-ci vous permet de partir du véhicule complet et de vous frayer un chemin jusqu'au plus minuscule des composants. En faisant appel à la structure `WITH RECURSIVE`, vous pourrez retrouver en peu de temps les spécifications de la vis platinée qui maintient l'électrode négative de la batterie auxiliaire.

La récursivité est également un procédé naturel pour résoudre un problème de type « Et si ? ». Dans l'exemple de Vannevar Airlines, et si la direction décide de ne plus desservir Charlotte depuis Portland, en quoi cela affectera-t-il la liste des villes que vous pouvez rejoindre depuis Portland ? Une requête récursive vous fournira rapidement la réponse.

Contrôler les opérations

DANS CETTE PARTIE :

Contrôler l'accès.

Protéger les données de la corruption.

Appliquer des langages procéduraux.

Chapitre 14
Protéger une base de données

DANS CE CHAPITRE :

- **Contrôler l'accès aux tables d'une base de données.**
- **Spécifier qui a accès à quoi.**
- **Accorder des privilèges d'accès.**
- **Retirer des privilèges d'accès.**
- **Empêcher les accès non autorisés.**
- **Transmettre le pouvoir d'accorder des privilèges.**

Tout administrateur système qui se respecte doit connaître les divers aspects du fonctionnement d'une base de données. C'est pourquoi j'ai traité dans les chapitres précédents des parties de SQL qui servent à créer et manipuler des bases de données, puis (au Chapitre 3) présenté les fonctions de SQL qui permettent de protéger les bases de données de divers sinistres. Dans ce chapitre, je vais examiner plus en détail les dommages susceptibles d'être engendrés par une mauvaise utilisation.

Le responsable en charge d'une base de données peut désigner les personnes qui y ont accès, mais aussi spécifier le niveau de leurs autorisations en leur accordant ou en leur retirant le droit d'accéder à certaines parties du système. L'administrateur système a même le pouvoir de transmettre (ou de retirer) une partie de ses pouvoirs en déléguant (ou refusant) à d'autres la possibilité d'accorder (ou non) certains privilèges à des utilisateurs. Pour peu que vous sachiez les utiliser correctement, les outils de SQL liés à la sécurité se révèlent particulièrement efficaces pour protéger des données importantes. Par contre, s'ils sont mal employés, ces mêmes outils peuvent empêcher des utilisateurs légitimes de faire leur travail.

Les bases de données contiennent souvent des informations destinées à rester confidentielles, ou du moins accessibles uniquement à des utilisateurs autorisés. C'est pourquoi SQL propose toute une palette de niveaux d'accès. Cela peut aller de tout à rien, avec différents degrés intermédiaires. Comme il a le pouvoir de contrôler avec précision les opérations auxquelles ont accès les utilisateurs, l'administrateur de la base de données peut leur accorder l'accès à toutes les données dont ils ont besoin pour travailler, tout en leur interdisant de voir ou de manipuler certaines parties de la base de données.

Le langage de contrôle de données de SQL (DCL)

Les instructions SQL que vous utilisez pour créer des bases de données forment le langage de définition de données (DDL, Data Definition Language). Une fois la base de données créée, vous pouvez utiliser un autre ensemble d'instructions SQL – connu sous le nom de langage de manipulation de données (DML, Data Manipulation Language) – pour ajouter, supprimer ou modifier des données. SQL comprend également une série d'instructions qui n'appartiennent à aucune de ces deux catégories. Les programmeurs s'y réfèrent quelquefois en les désignant collectivement sous le nom de langage de contrôle de données (DCL, Data Control Language). Les instructions DCL servent principalement à protéger la base de données contre les accès non autorisés, qu'ils trouvent leur origine dans une interaction fâcheuse entre plusieurs utilisateurs de la base ou une panne matérielle. Dans ce chapitre, je vais vous expliquer comment vous protéger de ces accès non autorisés.

Les niveaux d'accès de l'utilisateur

SQL permet de contrôler l'accès à neuf fonctions de gestion de la base :

- **Création, visualisation, modification, suppression :** Ces fonctions correspondent aux opérations `INSERT`, `SELECT`, `UPDATE` et `DELETE` dont j'ai parlé au Chapitre 6.

- **Référencement :** L'utilisation du mot clé `REFERENCES` (dont j'ai parlé aux Chapitres 3 et 5) consiste à appliquer des contraintes d'intégrité référentielle à une table qui dépend d'une autre table de la base.
- **Utilisation :** Le mot clé `USAGE` se rapporte aux domaines, aux jeux de caractères, aux interclassements et aux traductions (ces termes sont définis au Chapitre 5).
- **Définition de nouveaux types de données :** Vous gérez les noms de types de données définis par l'utilisateur via le mot clé UNDER.
- **Répondre à un événement :** L'utilisation du mot clé `TRIGGER` provoque l'exécution d'une instruction (ou d'un bloc d'instructions) SQL chaque fois qu'un événement prédéterminé survient.
- **Exécution :** Le mot clé `EXECUTE` provoque l'exécution d'un module de programme.

L'administrateur de la base de données

L'autorité suprême d'une base de données utilisée par plusieurs personnes est l'administrateur de la base de données (DBA, DataBase Administrator). Le DBA possède tous les droits et privilèges pour accéder à toutes les fonctionnalités de la base. Être DBA peut vous procurer un véritable sentiment de puissance, mais vous investit également d'une lourde responsabilité. Étant donné l'étendue de votre pouvoir, vous pouvez facilement semer le désordre dans votre base et détruire des milliers d'heures de travail. Un DBA doit toujours réfléchir clairement et soigneusement aux conséquences de ses actes.

Le DBA a tous les droits sur la base de données. Il contrôle de plus tous les droits des autres utilisateurs. De cette manière, des personnes hautement dignes de confiance peuvent avoir accès à des fonctions, et peut-être à des tables, interdites à la majorité des autres utilisateurs.

La meilleure façon de devenir DBA est d'installer vous-même un système de gestion de bases de données. Cette procédure vous donnera un compte, ou login, et un mot de passe. Ce compte vous identifie comme étant un utilisateur disposant de privilèges spéciaux. Le système désigne quelquefois cet utilisateur par DBA, quelquefois par administrateur système, voire par superutilisateur (désolé, le costume de super-héros n'est pas fourni par la maison). Dans tous les cas, votre première tâche, une fois que vous serez connecté, sera de modifier votre mot de passe par défaut pour le remplacer par quelque chose de plus secret.

Si vous ne changez pas le mot de passe par défaut, n'importe quelle personne qui lirait le manuel *serait en mesure d'usurper l'identité du DBA*. Je vous conseille de communiquer votre nouveau mot de passe uniquement à un petit nombre de personnes totalement dignes de confiance. Après tout, une météorite pourrait vous désintégrer demain matin, ou bien vous pourriez gagner au Loto et prendre le premier avion pour très loin au soleil, ou bien encore être dans l'impossibilité de venir au bureau pour une quelconque raison. Tout cela ne doit pas empêcher vos collègues de travailler. Une autre personne peut devenir à son tour administrateur système du moment qu'elle connaît le login et le mot de passe du DBA pour se connecter au système.

Si vous disposez des privilèges du DBA, je vous conseille de ne vous connecter sous cette identité que si vous devez effectuer une tâche qui requiert ce niveau de privilèges. Une fois ce travail accompli, déconnectez-vous. Puis reconnectez-vous en utilisant le login et le mot de passe d'un compte utilisé pour accomplir toutes les autres opérations de routine. Cette méthode pourra vous éviter de commettre des erreurs susceptibles d'avoir de fâcheuses conséquences pour les autres utilisateurs.

Les propriétaires des objets de la base de données

Une autre classe d'utilisateurs privilégiés, avec le DBA, est le propriétaire d'objets de la base de données. Par exemple, les tables et les vues sont des

objets de la base. Tout utilisateur qui crée de tels objets peut en définir le propriétaire. Le propriétaire d'une table bénéficie de tous les privilèges associés à celle-ci, y compris le droit d'autoriser d'autres utilisateurs à y accéder. Du fait que les vues s'appuient sur des tables sous-jacentes, une personne autre que le propriétaire d'une table a la possibilité de créer une vue basée sur cette dernière. Mais le propriétaire de cette vue ne pourra jamais exercer sur la table sous-jacente que les droits qui lui sont accordés. Rappelez-vous donc qu'un utilisateur ne peut jamais contourner la protection instaurée par le propriétaire d'une table par l'intermédiaire d'une simple vue.

Le public

Dans la terminologie réseau, le « public » désigne tous les utilisateurs qui ne sont pas spécialement privilégiés (autrement dit, ni les DBA ni les propriétaires d'objets) et à qui aucun utilisateur privilégié n'a accordé de droit particulier. Si un utilisateur privilégié accorde certains droits au public, toute personne pouvant accéder au système est investie de ces mêmes droits.

Dans la plupart des installations, il existe une hiérarchie des utilisateurs basée sur les privilèges qui leur sont accordés. Au sommet trône le DBA. Le public forme la base de la pyramide. La Figure 14.1 illustre cette hiérarchie.

Accorder des privilèges aux utilisateurs

Par la vertu de sa position, le DBA bénéficie de tous les privilèges sur tous les objets de la base de données. Après tout, le propriétaire d'un objet possède tous les privilèges portant sur cet objet, et la base de données est elle-même un objet. Par défaut, toute autre personne ne dispose d'aucun privilège sur aucun objet, à moins que quelqu'un qui possède déjà ces privilèges (ainsi que le droit de les transmettre) ne les lui ait explicitement transmis. Vous pouvez accorder des privilèges à un utilisateur en utilisant l'instruction `GRANT`. Sa syntaxe se présente de la manière suivante :

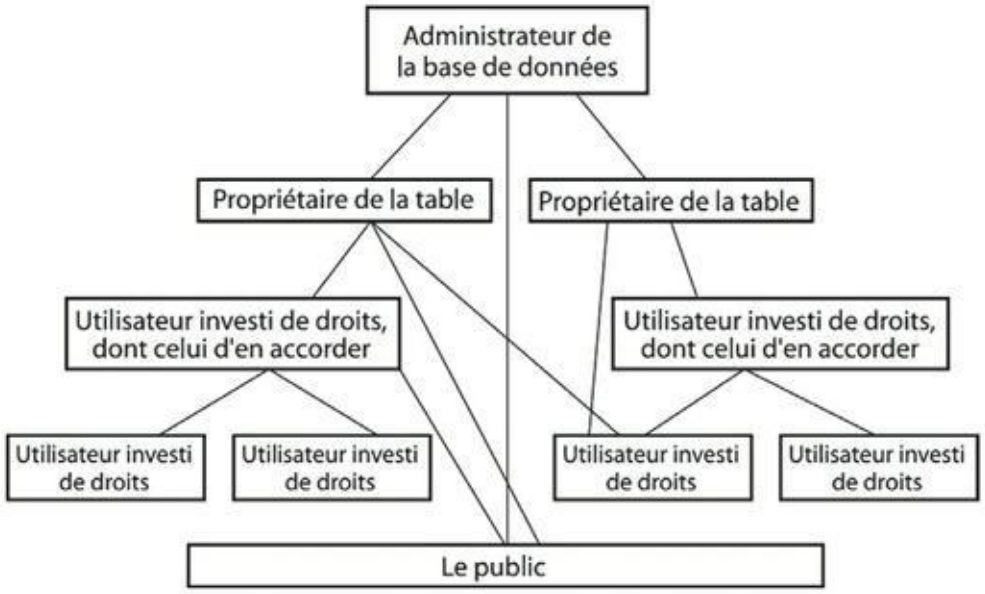


FIGURE 14.1 Hiérarchie des utilisateurs en fonction des privilèges d'accès.

```
GRANT liste_privilèges
ON objet
TO liste_utilisateurs
[WITH HIERARCHY OPTION]
[WITH GRANT OPTION]
[GRANTED BY auteur] ;
```

Dans cette instruction, *liste_privilèges* est défini ainsi :

```
privilège [, privilège]...
```

ou :

```
ALL PRIVILEGES
```

A son tour, *privilège* se définit comme suit :

```
SELECT
| DELETE

| INSERT [(nom_colonne[,nom_colonne]...)]
| UPDATE [(nom_colonne[,nom_colonne]...)]
```

```
| REFERENCES [(nom_colonne[,nom_colonne]...)]
| USAGE
| UNDER
| TRIGGER
| EXECUTE
```

Dans l'instruction originale, *objet* représente :

```
[TABLE] nom_table
| DOMAIN nom_domaine
| CHARACTER SET nom_jeu_de_caractères
| COLLATION nom_interclassement
| TRANSLATION nom_traduction
| TYPE nom_type_utilisateur
| SEQUENCE nom_générateur_séquence
| désignateur_routine_spécifique
```

Enfin, *liste_utilisateurs* se présente ainsi :

```
login [, login]...
| PUBLIC
```

L'*auteur* est soit `CURRENT_USER`, soit `CURRENT_ROLE`.

Dans la syntaxe précédente, les vues sont considérées comme étant des tables. Les privilèges `SELECT`, `DELETE`, `INSERT`, `UPDATE`, `TRIGGER` et `REFERENCES` ne concernent que les tables et les vues. Le privilège `USAGE` s'applique aux domaines, aux jeux de caractères, aux séquences d'interclassement et aux traductions. Le privilège `UNDER` regarde uniquement les types. Quant à `EXECUTE`, il s'applique en principe aux modules de programmes. Les sections suivantes présentent divers exemples d'utilisation de l'instruction `GRANT`.

Rôles

Un *nom d'utilisateur* est un type d'identificateur d'autorisation, mais ce n'est pas le seul. Il identifie une personne (ou un programme) qui est autorisée à effectuer une ou plusieurs actions sur la base de données. Dans une grande entreprise, accorder des privilèges particuliers à chaque employé peut vite se révéler fastidieux et coûteux en temps. SQL résout ce problème en introduisant la notion de rôle.

Un *rôle*, identifié par un nom de rôle, se compose d'un ensemble de zéros ou plusieurs privilèges pouvant être accordés à un ensemble d'individus ayant besoin du même niveau d'accès à la base de données. Par exemple, toutes les personnes qui ont le rôle `GARDE_SECURITE` doivent posséder les mêmes privilèges. Ces privilèges seront vraisemblablement différents de ceux accordés aux personnes qui ont le rôle de `VENDEUR_COMPTOIR`.

Les rôles ne font pas partie du noyau de la spécification de SQL. C'est pourquoi certaines implémentations ne les proposent pas. Reportez-vous à la documentation de votre SGBD avant de tenter de les utiliser.

Vous pouvez créer des rôles en utilisant la syntaxe suivante :

```
CREATE ROLE VENDEURS ;
```

Une fois le rôle créé, vous pouvez accorder des privilèges à des utilisateurs en faisant appel à l'instruction GRANT :

```
GRANT VENDEURS TO Charles ;
```

L'attribution de privilèges à un rôle s'effectue exactement comme avec les utilisateurs individuels. À une différence près : le rôle ne se plaint pas et il n'accuse pas le DBA de tous les maux.

Insérer des données

Pour accorder à un rôle le privilège d'ajouter des données dans une table, suivez cet exemple :

```
GRANT INSERT
ON CLIENTS
TO VENDEURS ;
```

Ce privilège permet à un vendeur d'ajouter de nouveaux clients dans la table CLIENTS.

Consulter des données

Pour permettre à un utilisateur de consulter les données d'une table, servez-vous de l'exemple suivant :

```
GRANT SELECT
ON PRODUITS
TO PUBLIC ;
```

Ce privilège permet à quiconque ayant accès au système (`PUBLIC`) de consulter le contenu de la table PRODUITS.

En réalité, accorder ce type de privilège est potentiellement dangereux. Les colonnes de la table PRODUITS peuvent contenir des informations confidentielles telles que `PRIX_DE_GROS`. Pour donner accès aux principales informations tout en interdisant la consultation des données sensibles, définissez une vue sur la table en éliminant les colonnes confidentielles. Accordez ensuite le privilège `SELECT` sur la vue et non sur la table. L'exemple suivant illustre la syntaxe à employer :

```
CREATE VIEW MARCHANDISES AS
SELECT MODELE, NOMPROD, DESCPROD, PRIX
          FROM PRODUITS ;
GRANT SELECT
ON MARCHANDISES
TO PUBLIC ;
```

En utilisant la vue MARCHANDISES, le public ne peut pas voir la colonne `PRIX_DES_PRODUITS` de la table PRODUIT ou toute autre colonne qui n'est pas mentionnée dans CREATE VIEW. Seules les colonnes explicitement définies dans la vue sont accessibles au public.

Modifier les données d'une table

Dans toute organisation active, les données stockées dans une table évoluent au fil du temps. Vous devez accorder à certaines personnes le droit et le pouvoir d'effectuer des modifications, tout en empêchant d'autres personnes de le faire. Pour accorder des privilèges de modification, suivez cet exemple :

```
GRANT UPDATE (PRIMESVNT)
ON PRIMES
TO RESPONSABLE_COMMERCIAL ;
```

Le responsable commercial peut ajuster le pourcentage des primes versées aux vendeurs (la colonne PRIMESVNT) en fonction des conditions du marché. Toutefois, il ne peut pas modifier les valeurs des colonnes `MONTANTMIN` et `MONTANTMAX` qui correspondent au niveau des primes. Pour autoriser la modification de toutes les colonnes, vous devez soit en lister tous les noms, soit n'en spécifier aucun, comme dans l'exemple suivant :

```
GRANT UPDATE
ON BONUS
TO DIRECTEUR_COMMERCIAL ;
```

Supprimer les lignes obsolètes d'une table

Des clients prennent leur retraite, changent d'activité ou choisissent un autre fournisseur. Des employés démissionnent, partent à la retraite, sont licenciés ou meurent. Des produits deviennent obsolètes. Bref, des informations dont vous conservez la trace peuvent perdre un jour leur intérêt. Quelqu'un doit donc supprimer les enregistrements devenus obsolètes. Bien entendu, déléguer ce droit demande une réflexion poussée. Là encore, l'instruction `GRANT` vous permet d'accorder un privilège de suppression à une ou des tierces personnes de la manière suivante :

```
GRANT DELETE
ON EMPLOYES
TO RESPONSABLE_DRH ;
```

Le directeur des ressources humaines peut supprimer des enregistrements de la table EMPLOYES, tout comme le DBA et le propriétaire de la table EMPLOYES (qui est probablement aussi le DBA). Aucune autre personne ne peut effacer des enregistrements (à moins qu'une autre instruction `GRANT` ne lui accorde ce pouvoir).

Référencer des tables liées

Si une table contient une clé étrangère qui correspond à la clé primaire d'une autre table, les données contenues dans la seconde table deviennent accessibles aux utilisateurs de la première. Cette situation crée potentiellement une dangereuse « porte de derrière » par laquelle des utilisateurs non autorisés pourraient passer pour extraire des informations confidentielles. Dans ce cas, en effet, un utilisateur n'a pas besoin de posséder des droits d'accès à une table pour découvrir quelque chose sur son contenu. Il lui suffit bien souvent d'avoir des droits d'accès sur une table qui référence cette cible pour pouvoir également accéder à cette dernière.

Par exemple, supposez que la table LICENCIES contienne les noms des employés qui vont être licenciés le mois prochain. Seuls les responsables autorisés ont un accès SELECT à la table. Mais voilà qu'un employé découvre que la clé primaire de la table est EMPID. Il crée alors une nouvelle table ESPION qui utilise `EMPID` comme clé étrangère, ce qui lui permet de jeter un coup d'œil sur `LICENCIES`. J'explique comment créer une clé étrangère avec une clause `REFERENCES` au Chapitre 5. Cette méthode mérite d'être placée en tête de la liste des techniques que tout administrateur système se doit de connaître. Voyons cela de plus près :

```
CREATE TABLE ESPION
 (EMPID INTEGER REFERENCES LICENCIES) ;
```

L'employé n'a plus qu'à insérer dans la table ESPION des lignes qui correspondent aux identifiants de tous les employés. Toute insertion qui échoue signifie que l'employé n'est pas licencié. Inversement, toute ligne que la table accepte d'ajouter signifie que la personne a du souci à se faire pour son avenir immédiat.

Tout n’est pas perdu. SQL permet de prévenir ce type d’intrusion en requérant d’un utilisateur privilégié qu’il accorde *explicitement* le droit de référencement aux autres utilisateurs. Par exemple :

```
GRANT REFERENCES (EMPID)
ON LICENCIEES
TO RESPONSABLE_DRH ;
```

Utiliser les domaines, les jeux de caractères, les séquences d’interclassement et les traductions

Les domaines, les jeux de caractères, les séquences d’interclassement et les traductions posent également des problèmes de sécurité. C’est plus particulièrement le cas des domaines. Vous devez absolument éviter qu’ils ne soient utilisés pour saper vos mesures de sécurité.

Vous pouvez définir un domaine qui regroupe un ensemble de colonnes. Vous voulez que toutes ces colonnes partagent le même type et les mêmes contraintes. Les colonnes que vous énumérez dans une instruction `CREATE DOMAIN` héritent des types et contraintes du domaine. Si vous le voulez, il reste possible de surcharger ces caractéristiques pour certaines colonnes en particulier. Pour autant, les domaines restent une solution élégante pour attribuer les mêmes caractéristiques à tout un ensemble de colonnes en une seule déclaration.

Recourir aux domaines se révèle pratique si vous utilisez une série de tables contenant des colonnes dont les caractéristiques sont identiques. Par exemple, la base de données de votre société peut comprendre plusieurs tables, chacune possédant une colonne `PRIX` qui doit être de type `DECIMAL (10,2)` et ne contenir que des valeurs positives inférieures à 10 000. Avant de créer vos tables, vous pourriez définir un domaine qui spécifie les caractéristiques de cette colonne de la manière suivante :

```
CREATE DOMAIN DOMAINE_TYPE_PRIX DECIMAL (10,2)
```

```
CHECK (VALUE >= 0 AND VALUE <= 10000) ;
```

Vous identifiez peut-être vos produits en les dénommant `CodeProduit` dans toutes vos tables. Ce code est toujours du type `CHAR (5)`, le premier caractère étant une lettre choisie dans la liste X, C, H, et le dernier l'un des deux chiffres 9 ou 0. Vous pouvez créer un domaine pour ces colonnes comme suit :

```
CREATE DOMAIN DOMAINE_CODE_PRODUIT CHAR (5)
CHECK (SUBSTR (VALUE, 1,1) IN ('X', 'C', 'H')
AND SUBSTR (VALUE, 5, 1) IN ('9', '0') ) ;
```

Les domaines étant en place, vous pouvez maintenant créer vos tables de la manière suivante :

```
CREATE TABLE PRODUITS
(CODE_PRODUIT DOMAINE_CODE_PRODUIT,
NOM_PRODUIT CHAR (30),
PRIX DOMAINE_TYPE_PRIX) ;
```

Au lieu de fournir explicitement le type de données pour `CODE_PRODUIT` et `PRIX` dans la définition de la table, vous spécifiez leur domaine, ce qui leur attribue le type adéquat et les contraintes explicitées dans l'instruction `CREATE DOMAIN`. Lorsque vous utilisez des domaines, vous devez les prendre en compte dans la gestion de la sécurité. Que se passe-t-il si une personne veut utiliser un domaine que vous avez créé ? Est-ce que cela peut poser des problèmes ? Que se passe-t-il si une personne crée une table dont une colonne a pour domaine `DOMAINE_TYPE_PRIX` ? Cette personne pourra tester progressivement des valeurs de plus en plus importantes sur la colonne jusqu'à ce qu'une entrée soit rejetée. Ce faisant, elle sera capable de déterminer la limite supérieure du type PRIX telle que vous l'avez spécifiée dans la clause `CHECK` de votre instruction CREATE DOMAIN. Si cette information a un caractère confidentiel, vous devez interdire aux autres l'accès à `DOMAINE_TYPE_PRIX`. Pour protéger vos tables de telles situations, SQL permet de limiter l'utilisation des domaines aux seules personnes explicitement autorisées par leur propriétaire. Seuls ces propriétaires (et le DBA) peuvent accorder une telle

permission. Si, après analyse, vous décidez que cet agrément ne mettra pas en péril votre sécurité, il vous suffira d'utiliser une instruction comme celle de l'exemple suivant :

```
GRANT USAGE ON DOMAINE_TYPE_PRIX TO
RESPONSABLE_COMMER-
CIAL ;
```

La suppression d'un domaine par `DROP` peut engendrer différents problèmes de sécurité. Si des tables contiennent des colonnes qui appartiennent au domaine que vous souhaitez supprimer, vous devrez sans doute détruire au préalable ces tables, sans quoi vous ne pourrez pas supprimer le domaine. Chaque implémentation gère la suppression d'un domaine d'une manière particulière. SQL Server la traite d'une certaine façon, Oracle d'une autre, et ainsi de suite. Dans tous les cas, il est plus prudent de restreindre les possibilités de suppression des domaines, tout comme celle des jeux de caractères, des séquences d'interclassement et des traductions.

Provoquer l'exécution d'instructions SQL

Il arrive parfois qu'une instruction SQL déclenche l'exécution d'une autre instruction, voire d'un bloc entier d'instructions.

Ce facteur déclenchant peut être activé par trois types de phénomènes : un événement, un instant spécifique ou une certaine action.

- Un **événement** peut déclencher (trigger) une certaine instruction ou un certain groupe d'instructions.
- Un **instant donné** peut déterminer le moment où une certaine action est déclenchée (soit juste avant, soit juste après l'événement).
- Une **action** correspond simplement à l'exécution d'une ou plusieurs instructions SQL.

Par exemple, vous pouvez utiliser un facteur déclenchant pour exécuter une commande qui vérifie la validité d'une nouvelle valeur avant d'autoriser un `UPDATE`. Si la nouvelle valeur n'est pas valide, la mise à jour n'aura pas lieu.

Un utilisateur ou un rôle doit posséder le privilège `TRIGGER` pour avoir le droit de déclencher une action. Par exemple :

```
CREATE TRIGGER SupprClient BEFORE DELETE
ON CLIENTS FOR EACH ROW
WHEN Deptt = '75
INSERT INTO LOGCLIENT VALUES ('Client parisien
suppri-
mé')  ;
```

Chaque fois qu'un client habitant Paris est sur le point d'être supprimé de la table CLIENTS, une entrée dans la table LOGCLIENT est ajoutée afin de mémoriser cet effacement.

Accorder des privilèges entre niveaux

Au Chapitre 2, je décris les types structurés comme une forme de types définis par l'utilisateur (UDT). L'essentiel de l'architecture des types structurés découle d'idées venant de la programmation orientée objet. L'une de ces idées est la *hiérarchie*, où un type peut avoir des *sous-types*, types dont certains des attributs dérivent de ceux du type dont eux-mêmes dérivent (leur *supertype*). En plus d'hériter de ces attributs, ils peuvent aussi disposer d'attributs qui leur sont propres. Une hiérarchie de cette nature peut comprendre de multiples niveaux, le type se trouvant tout en bas étant désigné comme un *type feuille*, par analogie à une arborescence.

Une table typée est une table dans laquelle chaque ligne est une instance du type structuré associé. Une telle table comprend une colonne par attribut de son type structuré associé. Le nom et le type de données de la colonne sont identiques à ceux de l'attribut.

Par exemple, supposons que vous soyez l'auteur de peintures que vous commercialisez dans des galeries. En plus de vos originaux, vous commercialisez aussi des tirages limités numérotés et signés, des tirages non numérotés, et des posters. Vous pouvez créer un type structuré pour vos œuvres de la manière suivante :

```
CREATE TYPE Soeuvres (
artiste CHARACTER VARYING (30),
titre CHARACTER VARYING (50),
description CHARACTER VARYING (256),
support CHARACTER VARYING (20),
dateCreation DATE )
NOT FINAL
```

C'est encore une fonctionnalité que tous les produits de SGBD ne proposent pas. Toutefois, PostgreSQL dispose de la commande `CREATE TYPE`, tout comme Oracle 11g et SQL Server 2012.

En tant qu'artiste souhaitant conserver la trace de vos œuvres, vous voulez faire la distinction entre les originaux et les reproductions. Vous pourriez de plus souhaiter distinguer les différents types de reproductions. La Figure 14.2 vous présente un exemple d'utilisation d'une hiérarchie pour faciliter ces distinctions. Le type Œuvre dispose de sous-types, qui peuvent à leur tour avoir des sous-types.

Il existe une correspondance un pour un entre les types dans la hiérarchie des types et les tables dans la hiérarchie de la table typée. Les tables standard, dont il est question aux Chapitres 4 et 5, ne peuvent pas figurer dans une hiérarchie semblable à celle évoquée ici pour les tables typées.

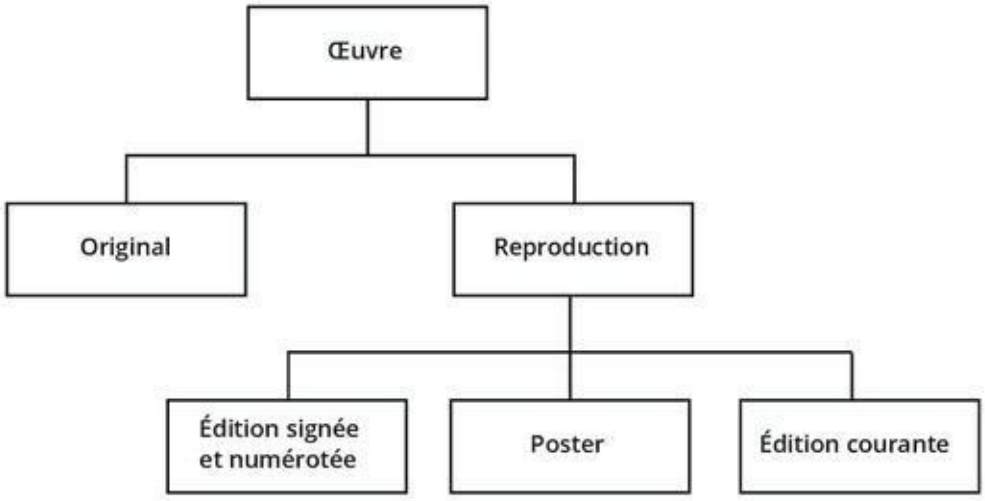


FIGURE 14-2 La hiérarchie des œuvres.

Au lieu d'utiliser une clé primaire, une table typée dispose d'une colonne d'auto-référence qui garantit l'unicité d'un enregistrement, non seulement dans la supertable qui occupe le sommet de la hiérarchie, mais dans toutes ses sous-tables. La colonne d'auto-référence est spécifiée par une clause `REF IS` dans la commande `CREATE` de la supertable. Lorsque ce système d'auto-référencement est mis en place, l'unicité est assurée dans toute la hiérarchie.

Donner le pouvoir d'accorder des privilèges

Le DBA peut accorder des privilèges à n'importe qui. Le propriétaire d'un objet peut accorder des privilèges sur cet objet à n'importe qui. Mais les utilisateurs bénéficiant de ces privilèges ne peuvent pas les rétrocéder à une tierce personne.

Seules les personnes désignées par le DBA ou le propriétaire peuvent bénéficier de ces privilèges. Histoire de maîtriser la situation...

Du strict point de vue de la sécurité, restreindre la délégation de privilèges est totalement justifié. Mais des utilisateurs ont souvent besoin de déléguer leur autorité. Le travail ne peut pas s'arrêter parce qu'une personne est malade, en vacances ou est sortie pour déjeuner.

Vous devez accorder à certains utilisateurs le pouvoir de transmettre leurs privilèges à d'autres personnes. Pour cela, utilisez l'instruction `GRANT` avec la clause `WITH GRANT OPTION`. Par exemple :

```
GRANT UPDATE (PRIMESVDT)
ON PRIMES
TO RESPONSABLE_COMMERCIAL
WITH GRANT OPTION ;
```

Le responsable commercial peut maintenant transmettre le privilège `UPDATE` en utilisant l'instruction suivante :

```
GRANT UPDATE (PRIMESVDT)
ON PRIMES
TO ASSISTANT_RESPONSABLE_COMMERCIAL ;
```

Une fois cette instruction exécutée, l'assistant du responsable commercial peut effectuer des modifications dans la colonne `PRIMESVDT` de la table PRIMES, ce qui lui était interdit auparavant.

Le fait de déléguer des droits représente toujours un compromis entre sécurité et commodité. Le propriétaire de la table PRIMES octroie un contrôle considérable en accordant le privilège `UPDATE` à l'assistant du responsable commercial via `WITH GRANT OPTION`. Ce propriétaire fait un pari sur le fait que son assistant prendra à cœur ses nouvelles responsabilités et qu'il ne courra pas lui-même le risque de déléguer à son tour ce privilège à n'importe qui.

Retirer des privilèges

Si vous avez le droit d'accorder des privilèges, vous devez également être en mesure de les retirer. Les postes occupés par les salariés de votre entreprise peuvent évoluer, et donc en même temps leurs besoins d'accès aux données. Il peut même arriver qu'un de vos employés quitte votre société pour aller travailler chez un concurrent. Dans ce cas, vous devez pouvoir lui retirer immédiatement tout privilège d'accès.

L'instruction REVOKE de SQL permet de supprimer des privilèges. Elle fonctionne comme l'instruction GRANT, à ceci près qu'elle a l'effet inverse. Sa syntaxe est la suivante :

```
REVOKE [GRANT OPTION FOR] liste_privilèges
ON objet
FROM liste_utilisateurs [RESTRICT|CASCADE] ;
```

Vous pouvez employer cette structure pour annuler des privilèges spécifiques tout en laissant intacts les autres. L'instruction REVOKE se distingue principalement de GRANT par la présence d'un des mots clés optionnels RESTRICT ou CASCADE.

Vous avez utilisé WITH GRANT OPTION pour accorder des privilèges à un utilisateur. Si vous voulez éventuellement annuler ces privilèges, utilisez l'option CASCADE dans l'instruction REVOKE. Vous supprimerez du même coup en cascade les privilèges que cet utilisateur a pu accorder à d'autres personnes.

D'un autre côté, l'option RESTRICT de l'instruction REVOKE ne fonctionne que si le délégataire n'a pas à son tour transmis ces droits spécifiques. En d'autres termes, vous retirez certains privilèges à tel utilisateur donné. Mais si celui-ci les a déjà diffusés à droite et à gauche, l'instruction REVOKE accompagnée de l'option RESTRICT ne révoquera personne et se contentera de retourner un message d'erreur.

Vous pouvez utiliser l'instruction REVOKE avec la clause GRANT OPTION FOR pour annuler le droit pour une personne de transmettre des privilèges (tout en les conservant pour elle-même).

Si la clause GRANT OPTION FOR et le mot clé CASCADE sont présents, vous révoquez tous les privilèges accordés à une personne donnée, de même que tous les droits de cet individu à rétrocéder ses privilèges. Dans le meilleur des cas, c'est en quelque sorte comme si vous n'aviez jamais rien autorisé. La présence simultanée de la clause GRANT OPTION FOR et du mot clé CASCADE peut produire l'une des situations suivantes :

- Si le bénéficiaire de l'instruction GRANT n'a jamais délégué les droits ainsi acquis à d'autres utilisateurs, l'instruction

REVOKE est exécutée, et elle supprime le ou les privilèges concernés en cascade.

- Si le bénéficiaire de l'instruction GRANT a déjà délégué au moins l'un des privilèges dont il dispose, l'instruction REVOKE n'est pas exécutée et elle renvoie un code d'erreur.

Le fait que vous puissiez accorder des privilèges avec WITH GRANT OPTION, puis que vous puissiez les révoquer sélectivement, rend la gestion de la sécurité plus complexe qu'il n'y paraît à première vue. Par exemple, plusieurs personnes ayant des droits identiques pourraient accorder le même privilège à un seul utilisateur. Si l'un de ces « délégataires » révoque ce privilège, l'utilisateur continuera à en bénéficier encore via les droits qui lui ont été accordés par d'autres. Si un privilège est transmis d'un utilisateur à un autre avec WITH GRANT OPTION, cette situation crée une chaîne de dépendance dans laquelle les privilèges d'un utilisateur dépendent de ceux d'un autre utilisateur. Si vous êtes DBA ou propriétaire d'un objet, veillez toujours, après avoir utilisé WITH GRANT OPTION, que le privilège ainsi délégué ne puisse pas réapparaître là où vous ne l'attendez pas. Supprimer un privilège pour certains utilisateurs indésirables, tout en le conservant pour ceux qui y ont légitimement droit, représente une tâche particulièrement délicate. En règle générale, l'utilisation des clauses GRANT OPTION et CASCADE engendre de nombreuses et délicates subtilités. Si vous utilisez ces clauses, lisez soigneusement la documentation de votre produit et les spécifications du standard SQL pour vous assurer que vous comprenez bien leur fonctionnement.

Utiliser simultanément GRANT et REVOKE pour gagner du temps et des efforts

Accorder de multiples privilèges sur des colonnes d'une table à de nombreux utilisateurs peut représenter des masses de lignes d'instructions. Considérez cet exemple : le directeur commercial souhaite que tous les

commerciaux puissent consulter la table CLIENTS. Mais seuls les responsables commerciaux peuvent mettre à jour, supprimer ou insérer des lignes. *Personne* ne devrait modifier le champ `CLIENT_ID`. Les noms des responsables commerciaux sont Tyson, Keith et David. Vous pouvez leur accorder les privilèges adéquats avec des instructions `GRANT` de la manière suivante :

```
GRANT SELECT, INSERT, DELETE
ON CLIENTS
TO TYSON, KEITH, DAVID ;
GRANT UPDATE
ON CLIENTS (SOCIETE, ADRESSE, VILLE,
ETAT, CODE_POSTAL, TELEPHONE)
TO TYSON, KEITH, DAVID ;
GRANT SELECT
ON CLIENTS
TO JENNY, VALERIE, MELODY, NEIL, ROBERT, SAM,
BRANDON, MICHELLE_T, ALLISON, ANDREW,
SCOTT, MICHELLE_B, JAIME, LINLEIGH, MATT, AMANDA
;
```

Ainsi chacun bénéficie des droits `SELECT` sur la table CLIENTS. Les responsables commerciaux bénéficient en plus des droits INSERT et `DELETE` complets sur cette table. Ils peuvent en particulier modifier toutes les colonnes à l'exception de `CLIENT_ID`.

La solution suivante donne plus rapidement le même résultat :

```
GRANT SELECT
ON CLIENTS
TO RESPVENTES ;
GRANT INSERT, DELETE, UPDATE
ON CLIENTS
TO TYSON, KEITH, DAVID ;
```

```
REVOKE UPDATE
ON CLIENTS (CUST_ID)
FROM TYSON, KEITH, DAVID ;
```

Le nombre d'instructions est identique, et le niveau de protection également. Mais ces instructions sont bien plus simples à saisir, car vous ne nommez pas tous les utilisateurs du département commercial ni toutes les colonnes de la table. Le temps de saisie ainsi gagné est également un gain de productivité, ce qui est l'idée sous-jacente principale.

Chapitre 15

Protection des données

DANS CE CHAPITRE :

- **Préserver une base de données.**
- **Comprendre les problèmes dus à des opérations concurrentes.**
- **Gérer les problèmes de concurrence avec les mécanismes de SQL.**
- **Protection sur mesure avec SET TRANSACTION.**
- **Protéger vos données sans nuire à la productivité.**

Vous devez connaître la loi de Murphy – en gros, si quelque chose peut aller de travers, elle ira de travers. Cette pseudo-loi est un sujet de plaisanterie, car, dans la plupart des cas, les choses se déroulent comme nous l'avions prévu. Si un problème inattendu survient, vous serez probablement en mesure de le détecter et de le résoudre.

Mais ce qui est vrai pour nous ne l'est pas forcément pour des structures complexes (un mathématicien vous expliquerait que le risque croît approximativement comme le carré de la complexité). Une base de données multi-utilisateur est une structure complexe. De multiples problèmes peuvent se produire lors de son fonctionnement. De nombreuses méthodes ont été développées pour minimiser l'impact de ces problèmes, mais elles n'ont jamais permis de les éliminer complètement (ce qui doit réjouir les professionnels de la maintenance des bases de données, car il semble virtuellement impossible d'automatiser leur travail). Dans ce chapitre, nous allons nous pencher sur les principaux risques qui menacent vos bases de données, et nous étudierons les outils que fournit SQL pour traiter les problèmes susceptibles de se poser à vous.

Menaces sur l'intégrité des données

Le cyberespace (y compris votre réseau) est un bel endroit à visiter, mais ce n'est pas tous les jours fête pour les données qui y résident. Vos données peuvent être endommagées ou corrompues de toutes sortes de manières. Au Chapitre 5, j'ai abordé les problèmes liés à une mauvaise saisie, aux erreurs de manipulation et aux destructions intentionnelles. Des instructions SQL mal formulées ou des applications mal conçues peuvent également endommager vos données. À ces menaces s'ajoutent l'instabilité de la plate-forme et les éventuelles pannes du matériel. Je vais traiter de ces deux derniers points dans les sections qui suivent, ainsi que des problèmes liés aux accès concurrents.

Instabilité de la plate-forme

L'instabilité de la plate-forme est une question qui ne devrait même pas exister. Malheureusement, elle se pose, et ce avec d'autant plus d'acuité que votre système fonctionne avec des composants qui n'ont pas été sérieusement testés. Un problème d'instabilité peut survenir après l'installation d'une nouvelle version du SGBD ou du système d'exploitation, ou encore après l'ajout d'un nouveau matériel. Vous êtes alors confronté à des situations totalement inattendues. Et cela se produit évidemment à un moment critique pour votre travail. Votre système se plante, vos données sont endommagées. Rien ne va plus. À part insulter votre ordinateur et ceux qui l'ont fabriqué, il ne vous reste guère d'autre solution que d'espérer que votre dernière sauvegarde soit la bonne.

Ne stockez jamais des données importantes sur un système dans lequel un composant quelconque n'a pas été testé. Résistez à la tentation de vous jeter sur la toute nouvelle version bêta, pourtant si alléchante, de votre SGBD ou de votre système d'exploitation. Si vous avez l'impression que vous devez vous faire la main sur un nouvel outil, faites-le sur une machine totalement isolée de votre environnement de production.

Panne matérielle

Le plus moderne des équipements n'est pas à l'abri d'une panne qui risque de faire franchir le Styx à vos données. Le risque n'est jamais nul, même avec un ordinateur dernier cri et haut de gamme. Si un incident matériel se produit alors que votre base de données est ouverte et active, vous pouvez d'évidence perdre des données, parfois même sans vous en rendre compte.

Ce type de panne survient toujours un jour ou l'autre. Et si ce jour-là la loi de Murphy s'applique, ce sera pour vous le pire des moments.

L'une des solutions pour protéger vos données des pannes matérielles est la *redondance*. Conservez une copie de tout. Pour une sécurité maximale, procurez-vous une copie conforme de la configuration matérielle de votre environnement de production (si tant est que cet investissement soit à la portée de votre organisation). Faites des sauvegardes de votre base de données et de vos applications que vous pourrez charger sur le double de votre environnement de production aussitôt qu'une panne se produit. Si le coût d'une telle redondance vous paraît exorbitant, faites a minima régulièrement une sauvegarde de votre base de données et de vos applications, de telle sorte que vous n'ayez pas à ressaisir une trop grande quantité de données en cas de panne.

Une autre solution pour vous préserver des pannes matérielles consiste à utiliser des transactions, dont il est question à la fin de ce chapitre. Une *transaction* est une unité indivisible de travail. Autrement dit, elle ne peut être exécutée qu'en entier ou pas du tout. Cette approche du tout ou rien peut paraître drastique. Cela dit, les pires problèmes apparaissent lorsqu'une série d'opérations sur la base est interrompue en cours de traitement. Les transactions diminuent donc le risque de voir vos données corrompues ou perdues, et ce même en cas de défaillance du matériel sur lequel réside la base.

Accès concurrent

Supposez que vous utilisiez un logiciel et du matériel certifiés, que vos données soient bonnes, que vos applications soient exemptes de bogues et que tout votre équipement soit parfaitement fiable. Le rêve ? Pas encore. Des problèmes peuvent apparaître lorsque plusieurs personnes essaient d'accéder simultanément à la même table d'une base de données (c'est l'accès concurrent), et que leurs ordinateurs tergiversent sur celui qui doit passer en premier (c'est la contention). Les systèmes de bases de données multi-utilisateurs doivent être capables de gérer efficacement une telle situation.

Problème d'interaction de transaction

Des problèmes de contention peuvent apparaître dans n'importe quelle application. Considérez le cas suivant : vous écrivez une application destinée à passer des commandes qui fait intervenir quatre tables : COMMANDES, CLIENTS, LIGNE_OB-JET et INVENTAIRE. Les conditions suivantes sont appliquées :

- La table COMMANDES a NUMERO_COMMANDE comme clé primaire et `NUMERO_CLIENT` comme clé étrangère qui référence la table CLIENTS.
- La table LIGNES_COMMANDE a `NUMERO_LIGNE` comme clé primaire et `NUMERO_OBJET` comme clé étrangère qui référence la table INVENTAIRE. Elle contient également une colonne QUANTITE.
- La table INVENTAIRE a `NUMERO_OBJET` comme clé primaire et contient également un champ appelé `QUANTITE_DISPO-NIBLE`.

Ces trois tables possèdent d'autres colonnes qui n'interviennent pas dans cet exemple. La stratégie de votre société consiste à traiter complètement chaque commande ou pas du tout. Autrement dit, aucune commande partielle n'est autorisée (pas de panique, ce n'est qu'une situation hypothétique). Vous allez écrire une application `TRAITER_COMMANDE` qui traitera chaque commande enregistrée dans la table COMMANDES. L'application va commencer par déterminer s'il est possible d'expédier tous les objets commandés. Si c'est le cas, elle écrira la commande, puis décrémentera comme il convient la colonne `QUANTITE_DISPONIBLE` de la table INVENTAIRE. Cette action supprimera au passage les entrées concernées des tables COMMANDES et LIGNES_COMMANDE.

L'application est conçue pour traiter les commandes de l'une des deux manières suivantes lorsque plusieurs utilisateurs accèdent simultanément à la base :

- La première méthode consiste à traiter la ligne d'INVENTAIRE qui correspond à chaque ligne de la table LIGNES_COMMANDE. Si `QUANTITE_DISPONIBLE` est

suffisant, l'application décrémente ce champ. Si la valeur de `QUANTITE_DISPONIBLE` n'est pas assez grande, elle annule la transaction et revient sur toutes les opérations qui ont affecté les autres objets LIGNES_COMMANDE de cette commande.

» La seconde méthode consiste à vérifier chaque ligne d'INVENTAIRE correspondant à une ligne LIGNES_COMMANDE de la commande. Si toutes les quantités sont suffisantes, l'application les traite en les décrémentant.

En règle générale, la première méthode s'avère la plus efficace si le traitement de la commande réussit. Par contre, la seconde méthode est plus efficace en cas d'échec. Si la plus grande part des commandes est honorée la plupart du temps, il est préférable d'employer la première méthode. À l'inverse, si une majorité de commandes a la fâcheuse habitude d'échouer, il vaut mieux appliquer la seconde méthode. Supposons que cette application hypothétique soit en cours d'exploitation sur un système multi-utilisateur dépourvu d'un contrôle adéquat des accès concurrents. Vous entendez déjà le sifflement des balles ? Voici un échantillon des problèmes qui peuvent alors survenir :

1. **Un client contacte un vendeur (nous l'appellerons utilisateur 1). Il veut commander 10 sécateurs (objet 1) et cinq tondeuses (objet 37).**

2. **L'utilisateur 1 emploie la méthode 1 pour traiter la commande. Il commence par les dix exemplaires de l'objet 1 (les sécateurs).**

   Comme il se doit, vous avez effectivement dix sécateurs en stock et l'utilisateur 1 les prend tous.

   La fonction de traitement des commandes décrémente la quantité en stock de l'objet 1 pour la passer à zéro. Et c'est

ici que les choses deviennent intéressantes. Un autre client contacte votre entreprise pour passer une (toute petite) commande. Cette fois, c'est l'utilisateur 2 qui lui répond.

3. **L'utilisateur 2 essaie d'enregistrer la commande d'un seul exemplaire de l'objet 1. Il constate alors qu'il ne reste plus aucun sécateur.**

   La commande de l'utilisateur 2 est annulée, car elle ne peut pas être honorée.

4. **Pendant ce temps, l'utilisateur 1 continue à traiter la commande en s'occupant des cinq exemplaires de l'objet 37 (les tondeuses).**

   Malheureusement, il n'y en a plus que quatre en stock. La commande de l'utilisateur 1 est donc annulée, puisqu'elle ne peut pas être honorée intégralement.

   La table INVENTAIRE se retrouve à présent dans son état initial, car aucun utilisateur n'a pu traiter sa commande. Les deux commandes sont perdues, bien que celle de l'utilisateur 2 aurait pu être satisfaite. Votre société est mal partie…

La méthode 2 n'est pas plus efficace, mais pour une raison différente. L'utilisateur 1 pourrait vérifier la disponibilité de tous les objets qu'il commande et décider que ces objets sont disponibles. Puis l'utilisateur 2 intervient et passe commande de l'un de ces produits avant que l'utilisateur 1 ait réalisé la décrémentation. La transaction de l'utilisateur 1 échoue.

Sérialiser pour éviter les problèmes d'interaction

Aucun conflit ne se produit si les transactions sont exécutées en série et non concurremment. Dans le premier exemple, si la transaction malheureuse de l'utilisateur 1 s'était terminée avant que ne débute celle de l'utilisateur 2, la fonction `ROLLBACK` aurait rendu disponible la tondeuse commandée par l'utilisateur 2. Rappelons que ROLLBACK est une machine à remonter le temps qui annule la totalité d'une transaction.

Si les transactions avaient été traitées en série dans le second exemple, l'utilisateur 2 n'aurait jamais pu modifier la quantité disponible d'un produit tant que la transaction de l'utilisateur 1 n'aurait pas été achevée. La transaction de l'utilisateur 1 se serait alors déroulée correctement (en se concluant soit par une réussite, soit par un échec), et l'utilisateur 2 aurait pu vérifier si l'objet qu'il souhaitait commander était réellement disponible.

Si les transactions sont exécutées en série, l'une après l'autre, elles n'interféreront jamais. L'exécution de transactions concurrentes est ditesérialisée si le résultat obtenu est le même que celui que produirait l'exécution en série de ces transactions.

Sérialiser des transactions concurrentes n'est pas la panacée. Entre se protéger des interactions malencontreuses et préserver les performances, il faut trouver le juste milieu. Plus vous isolez les transactions les unes des autres, plus il faut de temps pour les exécuter. Contrôler de manière trop tatillonne les accès concurrents peut nuire aux performances générales du système. Se protéger est indispensable, mais une surprotection ne fera que dégrader les choses.

Réduire le risque de corruption des données

Vous pouvez prendre des précautions à plusieurs niveaux pour éviter de perdre des données lors d'une interaction inattendue ou d'une panne. Il est possible de configurer votre SGBD de telle sorte qu'il gère certaines de ces précautions à votre place. Tel un ange gardien, il vous protégera du mal sans que vous n'en sachiez jamais rien. Vous ne verrez rien. D'ailleurs, vous ne voudrez sans doute même pas savoir que le SGBD est en train de vous aider. Votre administrateur de base de données (DBA) peut également établir des précautions à sa discrétion. Et vous en serez éventuellement informé ou non selon ce qu'il aura décidé. Enfin, en tant que développeur,

vous vous devez de suivre la même démarche lorsque vous écrivez votre code.

Vous vous éviterez un grand nombre de problèmes en prenant l'habitude d'appliquer automatiquement ces quelques principes élémentaires (que ce soit dans votre code ou dans vos interactions avec la base de données) :

- Utilisez les transactions SQL.
- Trouvez le juste milieu entre performances et protection.
- Sachez quand et comment utiliser des transactions, verrouiller des objets de la base de données et effectuer des sauvegardes.

Détaillons maintenant ces notions.

Utiliser les transactions SQL

La transaction est l'un des principaux outils qu'offre SQL pour maintenir l'intégrité d'une base de données. Une transaction SQL englobe toutes les instructions qui ont un effet sur la base. Elle se termine soit par une instruction `COMMIT`, soit par une instruction `ROLLBACK`.

- Si une transaction se termine par `COMMIT`, les effets de toutes les instructions de cette transaction sont répercutés dans la base de données en une seule séquence très rapide.
- Si une transaction se termine par `ROLLBACK`, les effets de toutes les instructions sont annulés et la base de données retrouve l'état qui était le sien avant que ne débute cette transaction.

Le terme application désigne ici soit l'exécution d'un programme (qu'il soit en Cobol, C++ ou tout autre langage), soit une série d'actions effectuées depuis un terminal lors d'une seule session.

Une application peut comprendre plusieurs transactions SQL. La première transaction commence avec cette application, et la dernière transaction se termine avec elle. Chaque `COMMIT` ou `ROLLBACK` effectué par l'application termine une transaction SQL et marque le début de la suivante. Par exemple, une application qui contient trois transactions SQL pourrait adopter la forme suivante :

```
Début de l'application
Diverses instructions SQL(SQL transaction-1)
COMMIT ou ROLLBACK
Diverses instructions SQL(SQL transaction-2)
COMMIT ou ROLLBACK
Diverses instructions SQL(SQL transaction-3)
COMMIT ou ROLLBACK
Fin de l'application
```

J'emploie le terme de « transaction SQL », car l'application peut faire appel à d'autres sortes de fonctionnalités de même type (par exemple lors d'un accès au réseau). Dans les pages suivantes, « transaction » sous-entend donc « transaction SQL ».

Une transaction SQL normale a un mode d'accès qui est soit `READ_WRITE` (lecture et écriture), soit `READ_ONLY` (écriture seule). Elle a un niveau d'isolation qui est soit SERIALIZABLE, soit `REPEATABLE READ`, soit `READ COMMITTED`, soit `READ UNCOMMITTED` (reportez-vous plus loin à la section « Niveaux d'isolation » pour plus de détails). Les caractéristiques par défaut sont `READ_WRITE` et `SERIALIZABLE`. Si vous voulez utiliser d'autres réglages, vous devez les spécifier avec une instruction SET TRANSACTION. Par exemple :

```
SET TRANSACTION READ ONLY ;
```

Ou encore :

```
SET TRANSACTION READ ONLY REPEATABLE READ ;
```

Ou bien :

```
SET TRANSACTION READ COMMITTED ;
```

Une application peut contenir plusieurs instructions `SET TRANSACTION`, mais vous ne pouvez en utiliser qu'une seule par transaction, et il doit s'agir de la première instruction exécutée dans le cadre de cette transaction. Il vous faut donc la placer soit au début de l'application, soit après `COMMIT` ou `ROLLBACK`. Vous devez faire appel à une instruction `SET TRANSACTION` au début de toute transaction qui ne se contentera pas des propriétés par défaut, car chaque nouvelle transaction suivant un `COMMIT` ou un `ROLLBACK` adopte automatiquement ces propriétés.

Une instruction `SET TRANSACTION` peut également spécifier un `DIAGNOSTICS SIZE` qui fixe le nombre de conditions d'erreur sur lesquelles l'implémentation devra conserver des informations (instaurer une telle limite est indispensable, car une implémentation peut détecter plus d'une erreur au cours d'une même instruction). La valeur SQL par défaut de cette limite dépend de l'implémentation, mais elle est pratiquement toujours adéquate.

La transaction par défaut

Les caractéristiques de la transaction SQL par défaut répondent en règle générale aux besoins de la plupart des utilisateurs. Mais vous pouvez spécifier d'autres propriétés via une instruction SET TRANSACTION comme cela a été expliqué dans la section précédente (il sera par ailleurs question de SET TRANSACTION plus loin dans ce chapitre).

La transaction par défaut s'appuie par ailleurs sur deux suppositions implicites :

- La base de données va changer au fil du temps.
- Il est toujours préférable de travailler en sécurité que de regretter ses actes.

C'est pourquoi elle applique le mode `READ-WRITE` qui vous permet d'exécuter des instructions destinées à modifier la base de données. De plus, elle utilise le niveau d'isolation `SERIALIZABLE`, qui correspond au

niveau le plus élevé d'isolation (et donc le plus sûr). `DIAGNOSTICS SIZE` prend une valeur par défaut qui dépend de l'implémentation. Reportez-vous à la documentation SQL de votre système pour en savoir plus.

Niveaux d'isolation

Dans l'idéal, le travail accompli par votre transaction est totalement isolé du travail accompli par les autres transactions, même si elles s'exécutent de manière concurrente. Mais, dans la réalité, il n'est pas toujours possible d'atteindre un niveau d'isolation parfait sur un système multi-utilisateur, à moins de sacrifier les performances. La question est donc : « Jusqu'à quel point souhaitez-vous isoler les transactions les unes des autres et combien êtes-vous prêt à payer pour cela en termes de performances ? »

Lecture sale

Le plus faible des niveaux d'isolation est appelé `READ UNCOMMITTED`. Il vous expose au problème de la lecture sale. Vous êtes en lecture sale lorsqu'un utilisateur peut consulter la valeur d'une donnée modifiée par une seconde personne avant que cette modification n'ait été validée par une instruction `COMMIT`.

Le problème apparaît lorsque le second utilisateur annule sa transaction. Les opérations effectuées ensuite par le premier utilisateur sont alors basées sur une valeur incorrecte. L'exemple classique est celui d'une application d'inventaire : un utilisateur décrémente l'inventaire, un second lit la nouvelle valeur. Le premier utilisateur annule sa transaction (ce qui rend à l'inventaire sa valeur initiale). Le second utilisateur, qui croit que le stock est presque épuisé, passe une commande pour le renouveler, alors qu'en réalité vous n'avez besoin de rien.

N'utilisez jamais le niveau d'isolation `READ UNCOMMITTED`, à moins que votre but ne soit pas de récupérer des résultats précis.

Vous pouvez vous placer au niveau `READ UNCOMMITTED` si vous voulez générer des statistiques telles que :

- Le délai maximal entre le passage de deux commandes.

- La moyenne d'âge des commerciaux qui n'atteignent pas leur objectif de vente.
- La moyenne d'âge des nouveaux employés.

Dans de tels cas, des informations approximatives suffisent. Le prix à payer en termes de performances pour un renforcement du contrôle des accès concurrents n'en vaut sans doute pas la chandelle (d'ailleurs, celle-ci ne vous aiderait pas à faire fonctionner votre réseau en cas de panne de courant...).

La lecture non répétable

Le niveau d'isolation suivant est `READ COMMITTED`. Une modification effectuée dans le cadre d'une autre transaction n'est pas vue par la vôtre tant que le second utilisateur n'a pas finalisé son opération en émettant un `COMMIT`. Ce niveau donne de meilleurs résultats que `READ UNCOMMITTED`, mais il pose le problème de la lecture qui ne peut être répétée. Cela ressemble à un titre de comédie, mais c'est un problème sérieux.

Reprenons l'exemple de l'inventaire. L'utilisateur 1 interroge la base de données pour connaître le nombre d'exemplaires disponibles d'un certain produit. La réponse est 10. Quasiment en même temps, l'utilisateur 2 entame puis valide une transaction qui passe une commande de 10 exemplaires de ce produit. Le stock est évidemment décrémenté, et passe donc à zéro. L'utilisateur 1, qui a vu que 10 exemplaires étaient disponibles, essaie d'en commander cinq. Mais il n'en reste même plus cinq. La première lecture de l'utilisateur 1 ne peut être répétée. La quantité a changé sans qu'il en soit averti. Toutes les décisions qu'il va prendre en fonction de sa première lecture sont invalides.

La lecture fantôme

Le niveau d'isolation `REPEATABLE READ` évite le problème précédent. Toutefois, ce niveau d'isolation est à son tour hanté par le problème de la lecture fantôme, qui survient lorsque la donnée que cet utilisateur est en train de lire est modifiée en réponse à une autre transaction et qu'il n'en est pas immédiatement informé à l'écran.

Supposons par exemple que notre utilisateur 1 passe une commande dont les critères de recherche (clauses `WHERE` et `HAVING`) désignent un ensemble de lignes. Immédiatement après, un utilisateur 2 modifie des données dans quelques-unes de ces lignes. Ces dernières répondaient peut-être aux critères de recherche de l'utilisateur 1 avant l'intervention de l'utilisateur 2, mais ce n'est plus le cas à présent. Il est même possible que des lignes qui ne vérifiaient pas les critères de recherche initiaux y répondent maintenant. L'utilisateur 1, dont la transaction est toujours active, n'est pas informé des modifications apportées par l'utilisateur 2. L'application se comporte comme si de rien n'était. Si l'utilisateur 1 effectue à nouveau la même recherche, il va récupérer des lignes différentes de celles qui lui avaient été retournées la première fois. La fiabilité des résultats a été vampirisée par la lecture fantôme.

Obtenir une lecture efficace (et sans doute plus lente)

Le niveau d'isolation `SERIALIZABLE` évite tous les problèmes décrits ci-dessus. À ce niveau, les transactions concurrentes peuvent être exécutées sans aucune interférence entre elles, et les résultats produits seront les mêmes que si elles avaient été exécutées en série (l'une après l'autre) et non en parallèle. Même à ce niveau d'isolation, des problèmes de logiciel et de matériel peuvent certes toujours provoquer l'échec de vos transactions. Mais vous n'aurez plus à vous inquiéter de la validité des résultats si vous êtes certain que votre système fonctionne correctement.

Naturellement, cette fiabilité supérieure a un prix en termes de performances. Le Tableau 15.1 fait le point sur les niveaux d'isolation et les problèmes qu'ils résolvent.

TABLEAU 15.1 Niveaux d'isolation et problèmes résolus.

Niveau d'isolation	Problèmes résolus	Lecture qui ne peut être répétée	Lecture qui ne peut être répétée	Lecture fantôme
`READ UNCOMMITTED`	Aucun	Non	Non	Non

READ COMMITTED	Lecture sale	Oui	Non	Non
REPEATABLE READ	Lecture sale	Oui	Oui	Non
SERIALIZABLE	Lecture sale	Non	Oui	Oui

Instruction implicite de début de transaction

Certaines implémentations de SQL exigent que vous leur signaliez le début d'une transaction à l'aide d'une instruction explicite telle que `BEGIN` ou `BEGIN TRAN`. Mais le standard SQL ne l'impose pas. Si vous n'êtes pas dans le cadre d'une transaction active, mais exécutez une instruction qui en requiert une, SQL initiera pour vous une transaction par défaut. `CREATE TABLE`, `SELECT` et `UPDATE` sont autant d'exemples d'instructions qui ont besoin pour fonctionner du contexte d'une transaction. Lancez une de ces instructions, et SQL ouvrira une transaction à votre place.

SET TRANSACTION

À l'occasion, vous pouvez avoir à utiliser une transaction dont les caractéristiques sont différentes des valeurs par défaut. Il est alors possible de spécifier ces propriétés à l'aide de l'instruction `SET TRANSACTION` (et ce, avant d'exécuter la première instruction qui requiert une transaction). `SET TRANSACTION` vous permet de spécifier un mode, un niveau d'isolation et une limite de diagnostic.

Pour changer par exemple ces trois caractéristiques à la fois, vous pouvez utiliser l'instruction suivante :

```
SET TRANSACTION
READ ONLY,
```

```
ISOLATION LEVEL READ UNCOMMITTED,
DIAGNOSTICS SIZE 4 ;
```

Avec ce paramétrage, vous ne pourrez pas exécuter d'instructions qui modifient la base de données (`READ ONLY`) et vous aurez spécifié le niveau d'isolation le plus bas (`READ UNCOMMITTED`). La limite de diagnostic est de 4. Vous utilisez donc le minimum des ressources du système.

Regardez maintenant cette instruction :

```
SET TRANSACTION
READ WRITE,
ISOLATION LEVEL SERIALIZABLE,
DIAGNOSTICS SIZE 8 ;
```

Ces caractéristiques vous permettent de modifier la base de données. Elles vous donnent le plus haut niveau d'isolation et repoussent la limite de diagnostic à la valeur 8. Ces paramètres mobilisent davantage les ressources du système. En fonction de la nature de l'implémentation que vous utilisez, ces propriétés peuvent (ou non) être identiques à celles de la transaction par défaut. Naturellement, toutes les combinaisons possibles sont a priori envisageables.

Utilisez toujours le niveau d'isolation qui correspond à vos besoins. L'utilisation constante du niveau `SERIALIZABLE` peut sembler une bonne solution, mais ce n'est pas vrai pour tous les systèmes. Selon votre implémentation et ce que vous faites, elle peut en effet pénaliser les performances de façon significative. Si vous ne comptez pas modifier votre base de données au cours de la transaction, passez au mode READ ONLY. Respectez la règle d'or : ne jamais s'approprier les ressources du système dont vous n'avez pas besoin.

COMMIT

Le standard SQL ne possède pas de mot clé explicite pour signaler le début d'une transaction. Par contre, il en contient deux qui spécifient la fin d'une transaction : `COMMIT` et `ROLLBACK`. Utilisez `COMMIT` si vous souhaitez valider les modifications apportées à la base de données. Vous disposez

aussi du mot clé optionnel `WORK` (`COMMIT WORK`). S'il se produit une erreur dans la base, ou si le système plante lors de l'exécution de `COMMIT`, vous devrez certainement annuler la transaction, puis essayer de la recommencer à nouveau.

ROLLBACK

Lorsqu'une transaction se termine, vous pouvez décider d'invalider les modifications apportées à la base lors de cette transaction. En fait, le but est alors de restaurer la base dans l'état qui était le sien avant que la transaction ne débute. Pour cela, exécutez une instruction `ROLLBACK`. `ROLLBACK` est un mécanisme résistant aux erreurs.

Même si le système se plante lors de l'exécution de ROLLBACK, vous pourrez relancer cette instruction une fois le système rétabli. Elle reprendra son travail et restaurera la base de données dans son état prétransactionnel.

Verrouiller les objets de la base de données

Le niveau d'isolation spécifié par défaut (ou par une instruction `SET TRANSACTION`) détermine le degré de zèle dont doit faire preuve votre SGBD pour protéger votre travail des interactions avec celui des autres utilisateurs. La principale protection que vous offre le SGBD contre des transactions dangereuses est la possibilité de verrouiller les objets de la base de données que vous utilisez. Voici quelques exemples :

UNE BASE ACID

Il arrive que certains concepteurs disent qu'ils veulent que leurs bases de données soient ACID. Cela ne signifie pas du tout qu'ils souhaitent les dissoudre dans un bouillon fumant. ACID est simplement un acronyme qui signifie Atomicité, Consistance, Isolation, Durabilité. Ces quatre caractéristiques sont nécessaires pour protéger une base de données contre la corruption :

- **Atomicité :** Une transaction devrait être atomique au sens classique du terme, autrement dit être traitée comme une unité indivisible. Soit elle est exécutée en totalité (`COMMIT`), soit la base est restaurée dans son état antérieur, comme si la transaction n'avait jamais existé (`ROLLBACK`).
- **Consistance :** Curieusement, la signification de la consistance n'est pas particulièrement cohérente (ou l'inverse). Elle varie en effet d'une application à une autre. Si vous transférez par exemple des fonds entre deux comptes bancaires, le total de l'argent sur l'ensemble des deux comptes ne devrait pas varier entre le début et la fin de la transaction. Dans une autre situation, votre critère de consistance pourrait très bien être différent.
- **Isolation :** Dans l'idéal, toutes les transactions qui s'exécutent simultanément sur une base de données devraient être totalement isolées les unes des autres. Cette condition est remplie si ces transactions sont sérialisables. Mais lorsqu'un système a à traiter un ensemble de transactions avec un très haut débit, abaisser le niveau d'isolation peut être utile pour améliorer les performances.
- **Durabilité :** Une fois votre transaction terminée (soit par `COMMIT`, soit par `ROLLBACK`), vous devez avoir la certitude que la base se trouve dans l'état voulu : avec des données parfaitement enregistrées, non corrompues, fiables, à jour. Même si votre système vous lâche brutalement après un `COMMIT` (mais avant que les résultats de la transaction soient enregistrés sur le disque), un SGBD durable doit être en mesure de garantir que la base sera bien restaurée dans l'état qui vient d'être décrit.

- La ligne de la table sur laquelle vous êtes en train de travailler est verrouillée. Les autres utilisateurs ne peuvent pas accéder à cet enregistrement.
- Une table est verrouillée dans sa totalité si vous effectuez une opération susceptible de l'affecter globalement.
- L'accès en lecture est autorisé, mais pas l'écriture. Parfois, c'est l'inverse.

Chaque implémentation dispose d'un système de verrouillage qui lui est propre. Certaines sont plus sûres que d'autres. Mais la majorité des systèmes récents vous protègent des problèmes les plus graves susceptibles de survenir lors d'accès concurrents.

Sauvegarder vos données

Votre DBA devrait effectuer régulièrement des sauvegardes. Il est indispensable de sauvegarder tous les éléments du système au bout d'un certain laps de temps qui est fonction de la fréquence de leur utilisation. Si vous mettez à jour quotidiennement votre base de données, il faut la sauvegarder tous les jours. Vos applications, formulaires et rapports peuvent aussi évoluer, mais moins souvent. Chaque fois qu'ils seront actualisés, votre DBA devra en sauvegarder les nouvelles versions.

Conservez plusieurs générations de sauvegarde. Un problème survenu dans la base de données peut n'être diagnostiqué que bien plus tard. Pour restaurer la dernière bonne version, vous devrez revenir en arrière de plusieurs sauvegardes.

Il existe plusieurs méthodes de sauvegarde :

- Utiliser SQL pour créer des tables de sauvegarde et y copier les données.
- Utiliser un mécanisme spécifique à l'implémentation qui sauvegarde tout ou partie du contenu de la base de

> données. Cette solution est en règle générale bien plus efficace et plus simple à mettre en œuvre que faire appel à SQL.

Votre installation peut comporter un mécanisme capable de tout sauvegarder, y compris les bases de données, les documents, les feuilles de calcul, les utilitaires et les jeux vidéo. Dans ce cas, il ne vous reste plus qu'à vérifier si les sauvegardes sont effectuées assez régulièrement pour vous protéger.

Points de sauvegarde et sous-transactions

Dans l'idéal, les transactions devraient être atomiques, aussi indivisibles que les Grecs le pensaient des plus petits éléments de la matière. Cependant, les atomes ne sont pas vraiment indivisibles. On a découvert depuis longtemps qu'ils sont constitués de parties plus petites, électrons, protons et neutrons, dont on sait aujourd'hui qu'elles ne sont même pas atomiques puisqu'elles sont constituées de quarks et de gluons.

Depuis la version SQL:1999, les transactions sur les bases de données ne sont pas non plus réellement atomiques. Une transaction peut être subdivisée en plusieurs sous-transactions. Chaque sous-transaction se termine par une instruction `SAVEPOINT`, qui est utilisée conjointement avec l'instruction `ROLLBACK`. Avant la création des points de sauvegarde (l'endroit du programme où l'instruction SAVEPOINT prend effet), l'instruction ROLLBACK était utilisée uniquement pour annuler l'intégralité de la transaction. À présent, vous pouvez en annuler seulement une partie.

La principale utilité de l'instruction `ROLLBACK` est de prévenir toute corruption des données quand une transaction est interrompue par une condition d'erreur. Annuler une transaction jusqu'à un certain point de sauvegarde n'a aucun sens si l'erreur s'est produite pendant le déroulement de la transaction. Dans ce cas, vous voulez évidemment restaurer la base de données dans l'état qui était le sien avant le début de cette transaction. Mais vous pouvez avoir d'autres raisons pour opérer une annulation partielle.

Supposons que vous effectuiez une série d'opérations complexes sur vos données. En cours de route, vous récupérez des résultats qui vous font

comprendre que vous vous êtes engagé dans une mauvaise voie. Si vous avez placé une instruction `SAVEPOINT` juste avant d'emprunter cette piste, vous pouvez annuler la transaction jusqu'au point de sauvegarde et essayer une autre direction. En supposant que le code était bon jusqu'au point de sauvegarde, cette solution sera bien plus efficace que d'effacer la totalité de la transaction pour tout recommencer à zéro.

Pour insérer un point de sauvegarde dans votre code SQL, utilisez la syntaxe suivante :

```
SAVEPOINT nom_pointsauve;
```

Vous pouvez déclencher l'annulation jusqu'à un point de sauvegarde à l'aide du code suivant :

```
ROLLBACK TO SAVEPOINT nom_pointsauve;
```

L'instruction SAVEPOINT n'est pas proposée dans toutes les implémentations. Vérifiez ce qu'en dit votre documentation.

Contraintes et transactions

Il ne suffit pas de vérifier que les données sont du bon type pour garantir qu'une base est valide. Par exemple, les valeurs nulles sont interdites dans certaines colonnes, tandis que d'autres doivent contenir uniquement des valeurs qui appartiennent à une certaine plage. J'aborde ces restrictions, également appelées contraintes, au Chapitre 5.

Les contraintes concernent naturellement les transactions, car elles peuvent vous empêcher de faire ce que vous voulez (après tout, elles sont d'ailleurs là aussi pour cela). Vous comptez par exemple ajouter des données à une table qui contient des colonnes sur lesquelles pèse la contrainte `NOT NULL`. Une méthode courante consiste à ajouter une ligne vide à la table, puis à y insérer plus tard des valeurs. Mais la contrainte `NOT NULL` qui pèse sur une colonne vous interdit d'employer cette stratégie. SQL ne permet pas d'ajouter une ligne qui contient une valeur nulle dans une colonne sur laquelle pèse une contrainte `NOT NULL`, et ce même si vous comptez y stocker une valeur avant la fin de votre transaction. Pour

contourner ce problème, SQL vous permet de désigner les contraintes comme étant soit DEFERRABLE, soit NOT DEFERRABLE.

Les contraintes NOT DEFERRABLE sont immédiatement appliquées. Vous pouvez spécifier qu'une contrainte DEFERRABLE est ou bien DEFERRED ou bien IMMEDIATE. Si une contrainte DEFERRABLE est IMMEDIATE, elle fonctionne comme une contrainte NOT DEFERRABLE (elle est donc appliquée immédiatement). Si une contrainte DEFERRABLE est DEFERRED, elle n'est pas appliquée tout de suite.

Pour ajouter des enregistrements vides ou effectuer toute autre opération qui peut violer des contraintes DEFERRABLE, vous pouvez utiliser une instruction comme celle-ci :

```
SET CONSTRAINTS ALL DEFERRED ;
```

Cette instruction change toutes les contraintes DEFERRABLE en condition DEFERRED sans affecter les contraintes NOT DEFERRABLE. Une fois que vous avez effectué toutes les opérations susceptibles de violer vos contraintes, et que la table atteint un état qui ne les viole pas, vous pouvez les appliquer à nouveau à l'aide de l'instruction suivante :

```
SET CONSTRAINTS ALL IMMEDIATE ;
```

Si vous avez fait une erreur et que l'une de vos contraintes est actuellement violée, vous le saurez dès l'exécution de cette instruction.

Si vous ne spécifiez pas explicitement que vos contraintes DEFERRED sont IMMEDIATE, SQL le fera à votre place lorsque vous essaierez de valider votre transaction par une COMMIT. Si une violation est détectée à ce moment, la transaction ne sera pas validée et SQL vous renverra un message d'erreur.

La gestion des contraintes SQL vous protège contre la saisie de valeurs invalides tout en vous permettant d'en violer temporairement certaines dans le cadre d'une transaction active.

L'exemple qui suit vous montre combien il est important de pouvoir différer l'application des contraintes.

Supposez qu'une table EMPLOYES contienne les colonnes EMP_ NO, EMP_NOM, DEPT_NO et SALAIRE. DEPT_NO est une clé étrangère qui référence la table DEPT. Cette table DEPT est formée des colonnes DEPT_NO et DEPT_NOM ; DEPT_NO est sa clé primaire.

En plus, vous voulez disposer d'une table semblable à DEPT qui contienne une colonne TOTALPAIE, afin d'y enregistrer le total des valeurs SALAIRE pour les employés de chaque département.

Vous pouvez créer l'équivalent de cette table sous forme d'une vue, comme ceci :

```
CREATE VIEW DEPT2 AS
SELECT D.*, SUM(E.SALAIRE) AS TOTALPAIE
          FROM DEPT D, EMPLOYES E
          WHERE D.DEPT_NO = E.DEPT_NO
          GROUP BY D.DEPT_NO ;
```

Il est également possible de définir cette vue ainsi :

```
CREATE VIEW DEPT3 AS
SELECT D.*,
          (SELECT SUM(E.SALAIRE)
                    FROM EMPLOYES E
                    WHERE D.DEPT_NO = E.DEPT_NO)
AS TO-
TALPAIE
          FROM DEPT D ;
```

Mais supposons maintenant que, pour des raisons d'efficacité, vous ne souhaitiez pas calculer le total chaque fois que vous référencez DEPT. TOTALPAIE. En réalité, vous voudriez plutôt ajouter une vraie colonne TOTALPAIE dans la table DEPT. Cette colonne serait mise à jour lors de chaque modification de salaire.

Pour vous assurer que la colonne SALAIRE est fiable, appliquez-lui une contrainte dans la définition de la table :

```
CREATE TABLE DEPT
(DEPT_NO CHAR(5),
DEPT_NOM CHAR(20),
TOTALPAIE DECIMAL(15,2),
CHECK (TOTALPAIE = (SELECT SUM(SALAIRE)
FROM EMPLOYES E WHERE E.DEPT_NO =
DEPT.DEPT_NO)));
```

À présent, vous voulez augmenter le SALAIRE de l'employé 123 de 100 unités. Vous allez par exemple utiliser l'instruction suivante :

```
UPDATE EMPLOYES
SET SALAIRE = SALAIRE + 100
WHERE EMP_NO = '123' ;
```

Vous ne devez pas non plus oublier d'actualiser la table DEPT :

```
UPDATE DEPT D
SET TOTALPAIE = TOTALPAIE + 100
WHERE D.DEPT_NO = (SELECT E.DEPT_NO
          FROM EMPLOYES E
          WHERE E.EMP_NO = '123') ;
```

(Vous utilisez une sous-requête pour référencer le DEPT_NO de l'employé 123.)

Mais il reste un problème. Les contraintes sont contrôlées après chaque instruction. En principe, toutes les contraintes sont vérifiées. Dans la pratique, les implémentations ne testent que les contraintes qui référencent des valeurs modifiées par l'instruction.

Après le premier UPDATE ci-dessus, l'implémentation vérifie toutes les références dont les valeurs sont modifiées par l'instruction, ce qui inclut celle qui est définie dans la table DEPT. En effet, cette contrainte référence la colonne SALAIRE de la table EMPLOYE et l'instruction UPDATE modifie cette colonne. La contrainte est violée après le premier UPDATE. Vous supposez que la base de données est dans un état correct avant

l'exécution de l'instruction UPDATE, et que chaque valeur TOTALPAIE de la table DEPT est égale à la somme des SALAIRE correspondants de la table EMPLOYES. Mais cette égalité n'est plus vraie dès que le premier UPDATE incrémente une valeur SALAIRE. La seconde instruction UPDATE corrige cette anomalie et remet la base de données dans un état où la contrainte est vérifiée. Mais entre les deux instructions, cette contrainte est bel et bien violée.

L'instruction SET CONSTRAINTS DEFERRED vous permet de désactiver ou de suspendre temporairement toutes ou certaines contraintes. La vérification de ces contraintes est différée jusqu'à l'exécution d'un SET CONSTRAINTS IMMEDIATE ou d'une instruction COMMIT ou ROLLBACK. Vous pouvez donc délimiter les deux UPDATE précédents par des instructions SET CONSTRAINTS. Le code se présente alors ainsi :

```
SET CONSTRAINTS DEFERRED ;
UPDATE EMPLOYES
SET SALAIRE = SALAIRE + 100
WHERE EMP_NO = '123' ;
UPDATE DEPT D
SET TOTALPAIE = TOTALPAIE + 100
WHERE D.DEPT_NO = (SELECT E.DEPT_NO
          FROM EMPLOYES E
          WHERE E.EMP_NO = '123') ;
SET CONSTRAINTS IMMEDIATE ;
```

Cette procédure diffère l'application de toutes les contraintes. Si vous insérez de nouvelles lignes dans DEPT, les clés primaires ne seront pas vérifiées. Vous venez donc de supprimer une protection dont vous pouvez avoir besoin. Par conséquent, il vaut mieux nommer explicitement les contraintes que vous voulez différer :

```
CREATE TABLE DEPT
 (DEPT_NO CHAR(5),
DEPT_NAME CHAR(20),
TOTALPAIE DECIMAL(15,2),
```

```
CONSTRAINT PAIE_CONTRAINTE
CHECK (TOTALPAIE = SELECT SUM(SALAIRE)
FROM EMPLOYES E
WHERE E.DEPT_NO = DEPT.DEPT_NO)) ;
```

Vous pouvez ensuite référencer individuellement vos contraintes :

```
SET CONSTRAINTS PAIE_CONTRAINTE DEFERRED ;
UPDATE EMPLOYES

SET SALAIRE = SALAIRE + 100
WHERE EMP_NO = '123' ;
UPDATE DEPT D
SET TOTALPAIE = TOTALPAIE + 100
WHERE D.DEPT_NO = (SELECT E.DEPT_NO
          FROM EMPLOYES E
          WHERE E.EMP_NO = '123') ;
SET CONSTRAINTS PAY_EQ_SUMSAL IMMEDIATE;
```

Si l'instruction `CREATE` ne contient aucun nom de contrainte, SQL en génère un implicitement. Ce nom implicite est stocké dans les tables du schéma (c'est-à-dire dans le catalogue). Toutefois, il vaut mieux employer des noms explicites. Supposez maintenant que, dans la seconde instruction UPDATE, vous avez spécifié par erreur une valeur d'incrément de 1 000. Cette valeur est autorisée dans l'instruction UPDATE, car l'application de la contrainte a été différée. Mais, lors de l'exécution de `SET CONSTRAINTS IMMEDIATE`, les contraintes spécifiées sont contrôlées. En cas d'échec, `SET CONSTRAINTS` provoque une exception. Si vous exécutez une instruction `COMMIT` à la place de `SET CONSTRAINTS IMMEDIATE` et que les contraintes ne sont pas évaluées comme étant toutes vérifiées, l'instruction `COMMIT` effectuera un `ROLLBACK`.

Rappelez-vous que vous ne pouvez différer l'application des contraintes que dans le cadre d'une transaction. Dès que cette dernière se termine soit par `ROLLBACK`, soit par `COMMIT`, les contraintes sont réactivées et vérifiées. Si vous utilisez correctement cette méthode, vous ne créerez pas

de données qui violeraient une contrainte disponible pour d’autres transactions.

Chapitre 16
Utiliser SQL dans des applications

DANS CE CHAPITRE :

- **Utiliser SQL dans une application.**
- **Combiner SQL avec des langages procéduraux.**
- **Faire face aux incompatibilités de langage.**
- **Intégrer SQL dans un code procédural.**
- **Appeler des modules SQL depuis un code procédural.**
- **Invoquer SQL depuis un outil RAD.**

Les précédents chapitres n'ont traité que de SQL. Des questions relatives à la manipulation des données ont été soulevées et des requêtes SQL développées pour y répondre. Cette méthode de travail, c'est-à-dire le SQL interactif, permet d'apprendre tout ce que SQL peut faire mais ne correspond pas du tout à la manière dont SQL est utilisé généralement.

Même si la syntaxe de SQL est proche de celle d'un langage naturel (en l'occurrence l'anglais), ce langage n'est pas facile à maîtriser. La grande majorité des utilisateurs d'ordinateurs ne sait toujours pas utiliser SQL. Et il est probable que cette situation perdurera, même si ce livre devient un best-seller. Dès qu'un utilisateur lambda se pose une question au sujet des bases de données, il ne pense certainement pas à s'asseoir devant son terminal pour saisir une instruction SELECT. Ce travail est réservé aux analystes et aux développeurs d'applications, lesquels ne font généralement pas carrière en saisissant des requêtes ad hoc dans des bases de données à longueur de journée. Ils développent des applications qui génèrent ces requêtes à leur place.

Si vous comptez effectuer une même tâche de manière répétitive, vous n'avez aucune envie de la saisir chaque fois via la console. Vous préférerez

écrire une application qui exécute cette tâche de sorte que vous puissiez l'exécuter immédiatement chaque fois que vous le souhaitez. SQL peut ainsi être intégré dans une application, mais il fonctionne alors d'une manière légèrement différente de celle que nous avons pu voir jusqu'à présent.

SQL dans une application

Au Chapitre 2, SQL est présenté comme un langage de programmation incomplet. Pour l'utiliser dans une application, vous devez le combiner à un langage procédural tel que Visual Basic, C, C++, C#, Java, COBOL ou encore Python. La structure particulière de SQL lui confère des avantages et des inconvénients. Les langages procéduraux, quant à eux, ont une structure différente, et donc différents avantages et inconvénients.

Fort heureusement, les avantages de SQL tendent à compenser les inconvénients des langages procéduraux. De même, les avantages des langages procéduraux résident justement là où SQL est faible. En combinant les deux, vous serez en mesure de créer des applications particulièrement efficaces. Il existe maintenant des environnements de développement d'application rapide orientés objet (RAD, Rapid Application Development), tels que Visual Studio de Microsoft ou l'environnement gratuit Eclipse, qui incorporent du code SQL dans vos applications via la manipulation d'objets plutôt que l'écriture de code procédural.

Gardez un œil sur l'astérisque

Dans les précédents chapitres, nous avons utilisé l'astérisque (*) comme notation abrégée signifiant « toutes les colonnes de la table ». L'astérisque est particulièrement utile si une table contient de nombreuses colonnes. Mais l'usage de ce caractère peut poser des problèmes lorsque vous intégrez du SQL dans une application.

Une fois l'application écrite, une tierce personne (ou vous-même) peut ajouter de nouvelles colonnes dans une table et/ ou en supprimer d'anciennes. Cela peut changer la signification « de toutes les colonnes ». L'application qui utilise l'astérisque peut alors récupérer des colonnes bien différentes de celles qu'elle attendait.

Ce type de modification d'une table n'affecte pas les programmes existants tant qu'ils n'ont pas été recompilés soit pour corriger un bogue, soit pour en

modifier certaines fonctionnalités (ce qui peut ne se produire que plusieurs mois plus tard). L'effet du caractère « * » est alors étendu à toutes les colonnes actuelles de la table, ce qui peut provoquer un plantage de l'application pour une tout autre raison que l'objet de la recompilation. Et le cauchemar commence...

Pour éviter ce type de problème, n'utilisez pas l'astérisque et spécifiez explicitement le nom de toutes les colonnes dans l'application.

Forces et faiblesses de SQL

SQL est parfaitement adapté à la récupération des données. Si des informations importantes sont enterrées quelque part dans une table d'une base de données, SQL vous fournira tous les outils dont vous avez besoin pour les retrouver. Vous n'avez pas besoin de connaître l'ordre des lignes et des colonnes dans une table, car SQL ne manipule pas ces objets individuellement. Les fonctionnalités transactionnelles de SQL vous garantissent que vos opérations sur la base de données ne seront pas affectées par des accès simultanés aux tables que vous êtes en train de manipuler.

Par contre, l'un des principaux inconvénients de SQL est la rusticité de son interface utilisateur. SQL ne propose rien pour présenter agréablement des écrans de saisie ou des rapports. Il accepte des commandes en mode console et retourne des valeurs sur l'écran du terminal ligne après ligne.

Parfois, une force dans un certain contexte peut s'avérer une faiblesse dans un autre. L'un des avantages de SQL est sa capacité à opérer sur une table entière en une fois. Une seule instruction `SELECT` peut traiter toutes les données d'une table, que celle-ci contienne une ligne ou des centaines, voire des milliers de lignes. Cependant, SQL n'est pas très à l'aise lorsqu'il s'agit de manipuler une ligne en particulier, ce qui peut constituer un problème si c'est ce que vous souhaitez faire. Dans ce cas, vous pouvez utiliser les curseurs de SQL ou faire appel à un langage procédural hôte.

Forces et faiblesses des langages procéduraux

Contrairement à SQL, les langages procéduraux sont conçus pour réaliser des opérations ligne par ligne, ce qui permet au développeur de l'application de contrôler très précisément le traitement appliqué à une table. Ce contrôle fin est une des grandes forces des langages procéduraux. Toutefois, c'est aussi l'un de leurs principaux inconvénients. Le développeur doit en effet savoir exactement où sont stockées les tables dans la base de données. L'ordre des lignes et des colonnes est important et doit être pris en compte.

Du fait de leur nature « étape par étape », les langages procéduraux offrent une grande souplesse quant à la création d'écrans de saisie et de visualisation des données. Ils vous permettent également de procéder à des impressions sophistiquées de rapports.

Problèmes liés à la combinaison de SQL et d'un langage procédural

Étant donné que les forces de SQL compensent les faiblesses des langages procéduraux et vice versa, il paraît cohérent d'associer les deux pour additionner leurs avantages sans être pénalisé par leurs faiblesses respectives. Mais aussi intéressante que soit cette combinaison, elle est loin d'être simple à mettre en œuvre. Vous devrez surmonter certains challenges avant d'en arriver au mariage parfait.

Différence des modes opératoires

Un des grands problèmes posés par la combinaison de SQL et d'un langage procédural est que SQL manipule des tables entières tandis qu'un langage procédural traite une ligne à la fois. Parfois, ce problème est mineur. Vous pouvez différencier les opérations qui s'appliquent à des tables de celles qui concernent des lignes en faisant appel aux outils appropriés.

Mais vous risquez d'avoir un problème si vous voulez rechercher dans une table les enregistrements qui répondent à certains critères, puis traiter ces enregistrements de différentes manières selon qu'ils répondent ou non aux conditions. Un tel processus requiert à la fois la puissance de SQL pour récupérer les données et les fonctionnalités du langage procédural pour traiter les conditions. Vous pouvez simplement intégrer des instructions SQL à des emplacements stratégiques d'un programme que vous avez écrit dans

un langage procédural (nous allons revenir un peu plus loin sur cette question).

Incompatibilités des types de données

L'intégration de SQL dans un langage procédural est confrontée à un autre problème : les types de données de SQL sont différents de ceux qui sont utilisés par les langages procéduraux les plus répandus. Cela n'a rien en soi de très surprenant, puisque les types de données diffèrent de toute manière d'un langage procédural à un autre.

Il n'existe aucune standardisation des types de données pour les langages de programmation. Avant la version 92 de SQL, l'incompatibilité des types de données posait un problème majeur. Mais depuis SQL-92 (et ses successeurs), l'instruction `CAST` permet de résoudre cette difficulté. Le Chapitre 9 explique en détail comment vous pouvez utiliser `CAST` pour convertir des types de données d'un langage procédural en un type reconnu par SQL, pourvu qu'il y ait compatibilité entre la source et la destination.

Utiliser SQL dans des langages procéduraux

Même si l'intégration de SQL dans des langages procéduraux se heurte à de nombreuses difficultés, elle n'en reste pas moins possible. En fait, vous devez même procéder de cette manière dans de nombreux cas, dès lors qu'il vous faut produire les résultats voulus dans le temps imparti. Vous disposez même pour cela de plusieurs méthodes. La section suivante en propose trois : SQL intégré, le langage de module et les outils RAD.

SQL intégré

La solution adoptée pour combiner du code SQL et du code écrit dans un langage procédural est le SQL intégré. Vous vous demandez comment cela peut fonctionner ? C'est aussi simple que ce que vous lisez : les instructions SQL sont parachutées en plein milieu du langage procédural là où il y en a besoin.

Comme vous le savez, une instruction SQL qui surgit soudainement au beau milieu d'un programme C peut poser un problème au compilateur qui ne s'y attend pas. C'est pourquoi les programmes qui contiennent du SQL intégré sont généralement traités par un préprocesseur avant d'être compilés ou interprétés. Le préprocesseur est averti de la présence de code SQL par la directive `EXEC SQL`.

Examinons un programme écrit dans la version Pro*C d'Oracle du langage C. Cette petite application accède à la table des employés d'une société, demande à l'utilisateur de saisir un nom, puis affiche le salaire et la commission de cet employé. Elle propose alors à l'utilisateur de saisir de nouvelles valeurs pour le salaire et la commission, après quoi elle met à jour la table avec les valeurs actualisées :

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR uid[20];
VARCHAR pwd[20];
VARCHAR enom[10];
FLOAT salaire, comm;
SHORT salaire_ind, comm_ind;
EXEC SQL END DECLARE SECTION;
main()
{
int sret; /* code retour de scanf */
/* Connexion */
strcpy(uid.arr,"FRED"); /* Copier le nom de
l'utilisateur
*/
uid.len=strlen(uid.arr);
strcpy(pwd.arr,"TOWER"); /* Copier le mot de
passe */
pwd.len=strlen(pwd.arr);

EXEC SQL WHENEVER SQLERROR STOP;
EXEC SQL WHENEVER NOT FOUND STOP;
EXEC SQL CONNECT :uid;
```

```
printf(“Utilisateur connecté : \n“ , uid.arr);
printf(“Saisissez le nom de l’employé à modifier
: “);
scanf(“%s”,ename.arr);
ename.len=strlen(ename.arr);
EXEC SQL SELECT SALAIRE,COMM INTO :salaire,:comm
          FROM EMPLOYES
          WHERE ENOM=:enom;
printf(“Employe: pourcentages salaire: %6.2f
comm: %6.2f
\n”,
          enom.arr, salaire, comm);
printf(“Saisissez le nouveau salaire : “);
sret=scanf(“%f”,&salaire);
salaire_ind = 0;
if (sret == EOF ! ! sret == 0) /* Positionner
l’indicateur
*/
salaire_ind =-1; /* Positionner l’indicateur pour
NULL */
printf(“Saisissez la nouvelle commission : “);
sret=scanf(“%f”,&comm);
comm_ind = 0; /* Positionner l’indicateur */
if (sret == EOF ! ! sret == 0)
comm_ind=-1; /* Positionner l’indicateur pour
NULL */
EXEC SQL UPDATE EMPLOYES
          SET SALARIE=:salaire:salaire_ind
          SET COMM=:comm:comm_ind
          WHERE ENOM=:enom;
printf(“Employé %s modifié. \n”,enom.arr);
EXEC SQL COMMIT WORK;
exit(0);
```

```
}
```

Nul besoin d'être un expert en C pour comprendre à quoi sert ce code et comment il le fait. Résumons tout de même le déroulement des instructions ci-dessus :

1. SQL commence par déclarer des variables hôtes.
2. Le code C contrôle la séquence de connexion de l'utilisateur.
3. SQL met alors en place la gestion d'erreurs et se connecte à la base de données.
4. Le code C demande le nom d'un employé et le stocke dans une variable.
5. Une instruction SQL `SELECT` récupère le salaire et la commission de l'employé ainsi désigné et les stocke dans les variables hôtes `: salaire` et `: com`.
6. Le code C reprend la main et affiche le nom de l'employé, son salaire et sa commission, puis demande à l'utilisateur de saisir les nouvelles valeurs du salaire et de la commission. Il vérifie que la saisie a été effectuée. Si elle ne l'a pas été, il positionne un indicateur.
7. Le SQL met à jour la base de données avec les nouvelles valeurs.
8. Le C affiche alors le message indiquant que l'opération est terminée.
9. Enfin, le SQL valide la transaction et le C met fin au programme.

Le préprocesseur vous permet de combiner des commandes des deux langages. Il sépare les instructions SQL des instructions du langage hôte en

les stockant dans une routine externe. Chaque instruction SQL est remplacée par un appel (CALL) du langage hôte vers cette routine externe. Le compilateur du langage peut alors faire son travail.

La manière dont le code SQL est transmis à la base de données dépend de l'implémentation. En tant que développeur, vous n'avez pas à vous préoccuper de cette question. C'est le préprocesseur qui s'en charge.

Déclarer des variables hôtes

Le programme écrit dans le langage hôte et les segments SQL ont besoin d'échanger des informations. Pour cela, il vous faut utiliser des variables hôtes que vous devez déclarer avant d'y faire appel, sans quoi SQL ne les reconnaîtra pas. Ces déclarations sont incluses dans un segment déclaratif qui précède le segment programme. Elles sont annoncées par la directive suivante :

```
EXEC SQL BEGIN DECLARE SECTION ;
```

La fin du segment déclaratif est signalée par :

```
EXEC SQL END DECLARE SECTION ;
```

Toute instruction SQL doit être précédée de la directive SQL EXEC. La fin d'un segment déclaratif peut éventuellement être signalée par une directive de terminaison. En COBOL, cette directive de terminaison est `END-EXEC`. En C, il s'agit d'un point-virgule.

Convertir des types de données

En fonction du degré de compatibilité des types de données du langage hôte et ceux de SQL, vous allez recourir si nécessaire à `CAST` pour réaliser certaines conversions. Vous pouvez utiliser pour cela des variables hôtes qui ont été déclarées dans `DECLARE SECTION`. N'oubliez pas de faire précéder le nom de la variable hôte d'un deux-points (:) lorsque vous l'employez dans des instructions SQL. Par exemple :

```
INSERT INTO ALIMENTS
 (NOM, CALORIES, PROTEINES, GRAISSES,
```

```
HYDRATECARBONE)
VALUES
 (:nom,  :calories,  :protéines,  :graisses,
:carbo) ;
```

Langage de module

Le langage de module est une autre solution pour utiliser SQL avec un langage procédural. Avec ce type de langage, vous stockez explicitement toutes vos instructions SQL dans un module distinct.

Un *module* SQL est tout simplement une liste d'instructions SQL. Chaque instruction est incluse dans une procédure SQL et est précédée d'une spécification du nom de la procédure ainsi que du nombre et du type de ses paramètres.

Chaque procédure SQL contient une seule instruction SQL. Dans le programme hôte, vous appelez explicitement la procédure là où vous comptez exécuter l'instruction SQL qu'elle contient. Tout se passe donc comme s'il s'agissait d'un sous-programme écrit dans le langage hôte.

L'utilisation de modules SQL revient en quelque sorte à effectuer à la main le travail du préprocesseur.

Le SQL intégré est bien plus souvent utilisé que la méthode « langage de module ». La plupart des éditeurs proposent une forme de langage de module, mais peu insistent sur le sujet dans leur documentation. Les langages de module présentent plusieurs avantages :

- **Les programmeurs en SQL n'ont pas besoin d'être en plus des spécialistes d'un langage procédural**. Comme SQL est complètement isolé du langage procédural, vous pouvez louer les services des meilleurs développeurs SQL pour écrire le code de vos modules (qu'ils soient ou non familiarisés avec votre langage procédural). En fait, il est même possible de différer le choix d'un langage procédural jusqu'à ce que vos modules SQL soient écrits et débogués.

- **Vous pouvez louer les services des meilleurs développeurs pour votre langage procédural, même s'ils n'y connaissent rien en SQL**. L'idée est simple : si vos technogourous SQL n'ont pas besoin d'être des experts du langage procédural, la réciproque est aussi vraie.
- **SQL n'est pas mélangé au code procédural, si bien que vous pouvez utiliser le débogueur de votre langage procédural**. Cela peut vous faire gagner un temps considérable de développement.

Encore une fois, un avantage vu sous un certain angle peut s'avérer un inconvénient d'un autre point de vue. Comme les modules SQL sont séparés du code procédural, il n'est pas si facile de suivre le fil logique de l'application lorsque vous essayez d'en comprendre le fonctionnement.

Déclarations dans un module

La syntaxe des déclarations dans un module à employer est la suivante :

```
MODULE [nom-module]
[NAMES ARE nom-jeu-caractères]
LANGUAGE
{ADA|C|COBOL|FORTRAN|MUMPS|PASCAL|PLI|SQL}
[SCHEMA nom-schéma]
[AUTHORIZATION id-autorisation]
[declarations-tables-temporaires...]
[declarations-curseurs...]
[declarations-curseurs-dynamiques...]
procédures...
```

Comme l'indiquent les crochets, le nom du module est optionnel. Toutefois, il est préférable de le nommer, ne serait-ce que pour éviter toute confusion dans le code.

La clause optionnelle NAMES ARE spécifie un jeu de caractères. Si vous ne spécifiez pas cette clause, c'est le jeu de caractères par défaut de votre implémentation SQL qui sera utilisé. La clause LANGUAGE indique au module quel est le langage qui l'appellera. Le compilateur doit disposer de cette information, car il doit faire croire au programme appelant que les instructions SQL ne sont qu'un sous-programme écrit dans son propre langage.

Bien que les clauses SCHEMA et AUTHORIZATION soient facultatives, vous devez spécifier au moins l'une d'entre elles (ou les deux). La clause SCHEMA spécifie le schéma par défaut et la clause AUTHORIZATION concerne l'identifiant de l'autorisation. Cet identifiant établit les privilèges dont vous disposez. Si vous ne précisez rien, le SGBD utilisera l'identifiant associé à votre session pour déterminer les privilèges attribués à votre module. Si vous n'avez pas les privilèges requis pour effectuer l'opération appelée par votre procédure, cette dernière ne sera pas exécutée.

Si votre procédure doit utiliser des tables temporaires, spécifiez-les avec une clause de déclaration de table temporaire. Déclarez de même les curseurs et les curseurs dynamiques avant toute procédure qui les utilise. Déclarer un curseur après une procédure est autorisé à partir du moment où ladite procédure n'utilise pas ledit curseur.

Les procédures du module

Enfin, après toutes ces déclarations se trouve une liste de procédures. Il s'agit de la partie fonctionnelle du module. Une procédure comporte un nom, des déclarations de paramètres et des instructions SQL exécutables. Le langage procédural appelle la procédure par son nom et lui passe des valeurs par l'intermédiaire des paramètres déclarés. La syntaxe d'une procédure est la suivante :

```
PROCEDURE nom-procédure
 (déclaration-paramètre [, déclaration-paramètre
]... )
SQL instruction;
 [SQL instructions ] ;
```

La déclaration d'un paramètre doit prendre la forme suivante :

```
nom-paramètre type-de-données
```

ou :

```
SQLSTATE
```

Les paramètres que vous déclarez peuvent être des paramètres d'entrée, des paramètres de sortie ou les deux à la fois. `SQLS-TATE` est un paramètre de statut servant à rapporter les erreurs.

Outils RAD orientés objets

Si vous utilisez des outils RAD évolués, vous pouvez développer des applications complexes sans avoir à écrire une ligne de code en C++, C#, Python, Java ou tout autre langage procédural. Il vous suffit de choisir des objets dans une librairie et de les placer visuellement à l'écran.

Les objets possèdent des propriétés qui les caractérisent et sont associés à des événements appropriés à leur type. Vous pouvez également associer une méthode à un objet. Une *méthode* est une procédure écrite dans un langage procédural. Mais vous pouvez construire des applications utiles sans écrire une seule méthode.

Bien que vous puissiez créer des applications complexes sans recourir à un langage procédural, vous aurez probablement tôt ou tard besoin de SQL. En effet, il est difficile, voire impossible, de reproduire la richesse d'expression de SQL dans un monde orienté objet. C'est pourquoi les outils RAD élaborés proposent un mécanisme pour injecter des instructions SQL dans vos applications orientées objet. Visual Studio de Microsoft en est un exemple. Access de Microsoft est un autre environnement de développement d'applications qui vous permet d'utiliser SQL en combinaison avec son langage procédural, VBA.

Le Chapitre 4 vous montre comment créer des tables de bases de données avec Access. Naturellement, cette opération ne représente qu'une faible partie des possibilités d'Access. Celui-ci est un outil qui sert avant tout à réaliser des applications destinées à traiter les données extraites des tables de vos bases de données. En utilisant Access, le développeur positionne des objets sur des formulaires, puis personnalise ces objets en leur donnant des propriétés, des événements et des méthodes. Ces formulaires et

méthodes sont ensuite manipulés par du code VBA capable d'intégrer des instructions SQL.

Bien que les outils RAD tels qu'Access puissent permettre de produire en peu de temps des applications de grande qualité, ils sont généralement spécifiques à une plate-forme. Par exemple, Access ne fonctionne que sous le système d'exploitation Windows de Microsoft. Ne l'oubliez pas si vous envisagez un jour d'écrire des fonctionnalités indépendantes du support, ou encore de migrer vos applications vers d'autres plates-formes.

Les outils RAD tels qu'Access sont les premiers fruits d'un possible rapprochement entre les approches relationnelles et orientées objet des bases de données. SQL et les avantages offerts par les structures relationnelles survivront tous deux. Ils bénéficieront des capacités de développement rapide (et comparativement moins sujet aux erreurs) apportées par la programmation orientée objet.

Utiliser SQL avec Microsoft Access

Microsoft Access cible essentiellement les personnes qui souhaitent développer des applications relativement simples sans se soucier particulièrement de programmation. Si tel est votre cas, vous trouverez dans la collection « Pour les Nuls » de quoi répondre à vos besoins ! Si vous voulez aller plus loin en associant le langage procédural d'Access (VBA) et SQL, la remarque précédente est encore plus vraie, tant la documentation livrée par Microsoft est limitée sur ce sujet. Mais attention : l'implémentation SQL que vous trouvez dans Access est loin d'être complète, et il vous faudra tout le flair d'un Sherlock Holmes pour la débusquer.

Dans le Chapitre 3, j'ai décrit les trois composants de SQL:le langage de définition de données (DDL), le langage de manipulation de données (DML) et le langage de contrôle des données (DCL). Le sous-ensemble de SQL contenu dans Access implémente uniquement la partie DML. Les opérations de création des tables sont réalisées à l'aide de l'outil RAD présenté au Chapitre 4. La même technique est employée pour l'implémentation des mesures de sécurité (elles sont traitées au Chapitre 14).

Pour voir un coin de SQL sous Access, il est nécessaire de soulever quelque peu le voile posé par le RAD. Prenons à titre d'exemple une base de données censée être gérée par une organisation non gouvernementale (fictive) : la SEL (Société pour l'écologie lunaire). Cette organisation

comprend plusieurs équipes de recherche, dont notamment la célèbre ERBALU (Equipe de recherche de la base lunaire). Une grave question se pose : quelles sont les communications savantes écrites par les membres de cette équipe ? Pour y répondre, une requête a été formulée en faisant appel au mode QBE d'Access (requête par l'exemple). Cette requête, qui est illustrée sur la Figure 16.1, récupère les informations voulues à partir des tables EQUIPES-RECHERCHE, AUTEURS et ARTICLES, avec l'aide de tables de recoupage (AUT_ART et AUT_RECH) qui ont été ajoutées afin de couper les relations plusieurs à plusieurs.

Après avoir cliqué sur l'onglet Accueil pour accéder à la barre d'outils, vous pouvez cliquer sur le menu déroulant de l'icône Vue dans l'angle supérieur gauche de la fenêtre pour révéler les autres types de vues disponibles lors de la création de requêtes. L'un des choix proposés est le mode SQL (voir la Figure 16.2). Lorsque vous activez ce mode, la fenêtre de l'éditeur SQL apparaît. Elle affiche l'instruction SQL générée par Access à partir des choix opérés dans l'interface graphique.

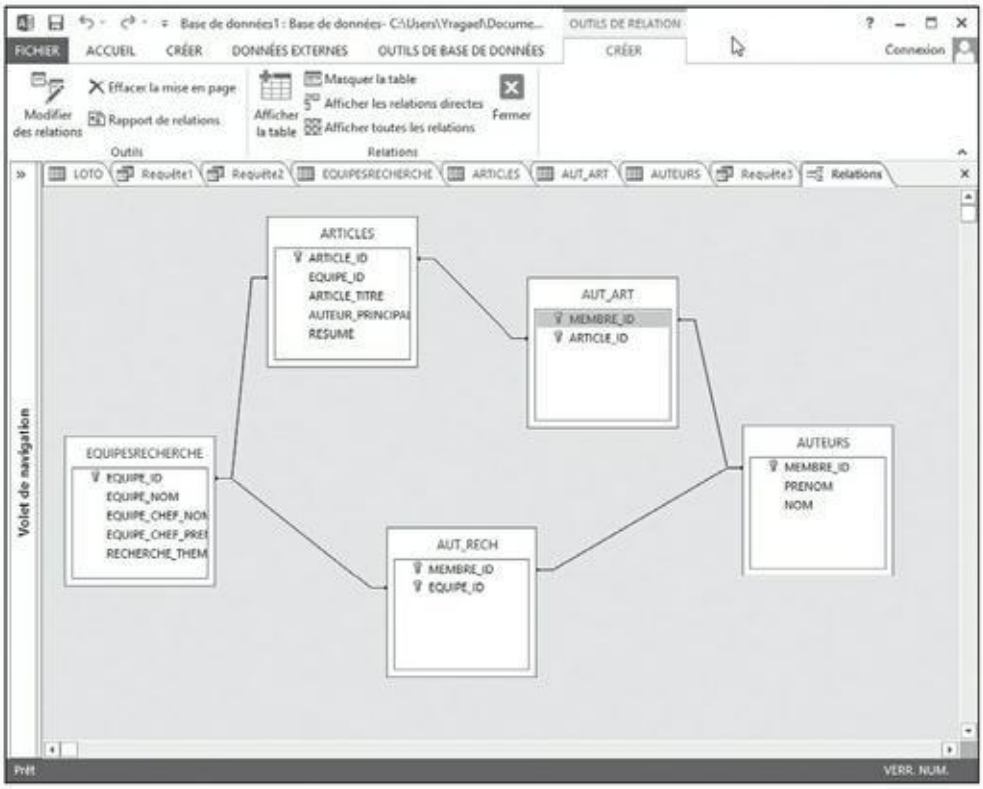


FIGURE 16.1 Conception de la requête Articles ERBALU en mode Création.

FIGURE 16.2 La vue SQL est proposée dans la liste des affichages.

L'instruction SQL montrée sur la Figure 16.3 est celle qui est effectivement envoyée au moteur de la base de données. Ce moteur, qui s'interface directement avec la base, ne comprend que le langage SQL. Tout ce que vous entrez dans le mode dit Création d'Access est traduit en SQL avant envoi pour traitement au moteur de la base de données.

FIGURE 16.3 Une instruction SQL qui sert à retrouver les noms de tous les articles écrits par des membres de l'équipe ERBALU.

Vous avez peut-être remarqué que la syntaxe de l'instruction illustrée sur la Figure 16.3 diffère quelque peu de celle du standard SQL ANSI/ISO. Souvenez-vous du proverbe : « Quand tu es à Rome, vis comme les Romains. » Lorsque vous travaillez avec Access, utilisez son dialecte SQL. Ce conseil vaut d'ailleurs quel que soit l'environnement dans lequel vous travaillez. Toutes les implémentations SQL divergent du standard sur tel ou tel aspect.

Si vous voulez écrire directement une requête dans le SQL d'Access (autrement dit, sans passer par le mode de création graphique), il vous suffit d'effacer le contenu courant de la fenêtre d'édition et de saisir une nouvelle instruction SELECT. Cliquez ensuite sur le bouton de la barre d'outils qui affiche un point d'exclamation pour exécuter votre requête. Le résultat va apparaître par défaut sous la forme d'une feuille de données.

Utiliser SQL dans le monde réel

DANS CETTE PARTIE :

Utiliser ODBC.

Utiliser JDBC.

Travailler sur des données XML.

Chapitre 17
ODBC et JDBC

DANS CE CHAPITRE :

- **Définition d'ODBC.**
- **Description des diverses parties d'ODBC.**
- **Utilisation d'ODBC dans un environnement client/serveur.**
- **Utiliser ODBC sur Internet.**
- **Utiliser ODBC sur un intranet.**
- **Utiliser JDBC.**

Ces dernières années, l'interconnexion des ordinateurs s'est massivement développée, aussi bien dans les organisations qu'entre elles. Cette connectivité frénétique a engendré le besoin de partager sur les réseaux les informations fournies par les bases de données. Le principal obstacle au libre partage de l'information sur les réseaux est l'incompatibilité des systèmes d'exploitation et des applications qui fonctionnent sur différentes machines. Cet obstacle a été en partie contourné par la création de SQL, puis par les évolutions qui l'ont fait progresser au fil des années.

Hélas ! Le SQL « standard » n'est pas véritablement un standard. Même les éditeurs de SGBD, qui se disent conformes au standard international SQL, ont inclus dans leur implémentation de SQL des fonctionnalités qui les rendent incompatibles avec les extensions propriétaires des implémentations d'autres éditeurs. Et tous rechignent à abandonner leurs propres extensions, car leurs clients les utilisent dans leurs applications et en sont devenus dépendants. Il faut donc trouver une autre voie pour assurer la communication entre SGBD, un outil qui n'oblige pas les éditeurs à « niveler par le bas » leurs implémentations jusqu'à atteindre un plus petit dénominateur commun. Cette solution, c'est ODBC (Open DataBase Connectivity).

ODBC

ODBC est une interface standard entre une base de données et une application qui accède au contenu de cette base. Recourir à ce standard permet à un front end applicatif d'accéder à n'importe quelle base de données en utilisant SQL. La seule condition est que front end et back end adhèrent tous deux au standard ODBC, dont la version courante est ODBC 4.0.

Une application accède à une base en utilisant un *pilote* (ou *driver*) spécifiquement conçu pour s'interfacer avec cette dernière. Le front end du pilote, c'est-à-dire la partie qui se connecte à l'application, se conforme scrupuleusement au standard ODBC. Il présente toujours la même interface à l'application, et ce quel que soit le moteur de la base de données qu'il permet d'attaquer. Le back end du pilote est conçu pour dialoguer avec un type spécifique de moteur de base de données. Grâce à cette architecture, les applications n'ont plus besoin d'être modifiées en fonction du moteur de base de données qui contrôle les données que ces applications manipulent. Toute la « machinerie » interne est masquée.

L'interface ODBC

L'interface ODBC est essentiellement un ensemble de définitions reconnu comme étant un standard. Ces définitions concernent tout ce qui doit être pris en compte pour établir une communication entre l'application et la base de données.

L'interface ODBC définit les éléments suivants :

- **Une librairie d'appels de fonction.**
- **Une syntaxe SQL standard.**
- **Des types de données SQL standard.**
- **Un protocole standard pour se connecter à un moteur de base de données.**
- **Des codes d'erreur standard.**

Les *appels de fonction* ODBC permettent de se connecter à un moteur de base de données, d'exécuter des instructions SQL et de renvoyer les résultats produits à l'application.

Pour effectuer une opération sur la base de données, vous devez transmettre en argument l'instruction SQL appropriée à un appel de fonction ODBC. Tant que vous utilisez la syntaxe SQL standard comprise par ODBC, l'opération fonctionne, indépendamment du moteur de base de données que vous attaquez.

Les composants d'ODBC

L'interface ODBC est composée de quatre couches fonctionnelles qui s'intercalent de la manière suivante entre l'utilisateur et les données sur lesquelles il souhaite travailler (le but étant d'assurer une communication transparente entre n'importe quel front end compatible et n'importe quel back end compatible) :

- **L'application** : C'est la partie de l'interface ODBC la plus proche de l'utilisateur. L'application doit être informée qu'elle communique avec une source de données via ODBC. Elle doit se connecter en douceur au pilote ODBC en respectant scrupuleusement ce standard.
- **Le gestionnaire du pilote** : Il s'agit d'une DLL (*Dynamic Link Library*) généralement fournie par Microsoft. Le gestionnaire charge les pilotes qui correspondent aux sources de données du système (il peut y en avoir plusieurs) et dirige les appels de fonction provenant de l'application vers les sources de données adéquates via les pilotes adaptés. Ce gestionnaire traite aussi directement certains appels de fonction ODBC. Enfin, il détecte et gère certains types d'erreurs.
- **Le pilote** : Comme les sources de données peuvent être différentes les unes des autres (et souvent très différentes),

vous devez trouver un moyen de traduire les appels de fonction ODBC dans le langage natif que comprend chaque source. Cette traduction est assurée par la DLL du pilote. Chaque DLL de pilote reçoit des appels de fonction via l'interface standard ODBC et les transcrit dans un code que sa *source de données* peut comprendre. Lorsque cette source retourne un résultat, le pilote le reformate dans l'autre sens afin de produite un résultat conforme au standard ODBC. Le pilote est l'élément clé qui permet aux applications compatibles ODBC de manipuler la structure et le contenu de sources de données elles-mêmes compatibles ODBC.

- **La source de données** : Elle peut être de nature différente. Ce peut être un SGBD relationnel (et sa base de données associée) qui réside sur le même ordinateur que l'application. Ce peut être également une base de données sur un ordinateur distant. Il peut aussi s'agir d'un fichier ISAM (Indexed Sequential Access Method, méthode d'accès séquentielle indexée) sans SGBD associé, et qui se trouve soit sur un ordinateur local, soit sur un système distant. Quelle que soit la forme que prend la source de données, vous devez disposer d'un pilote spécifique pour communiquer avec elle.

ODBC dans un environnement client/serveur

Dans un environnement client/serveur, l'interface entre le client et le serveur est appelée *API* (Application Programming Interface, interface de programmation d'application). Un pilote ODBC, par exemple, contient une

API. Une API peut être propriétaire ou standard. Dans le cas d'une API propriétaire, la partie client de l'interface a été spécifiquement conçue pour fonctionner avec un back end particulier sur le serveur. Le code qui constitue cette interface est un pilote, également appelé *pilote natif* sur un système propriétaire. Un pilote natif est optimisé pour ne fonctionner qu'avec un client spécifique et sa source de données associée. Du fait de cette spécialisation, les pilotes natifs ont tendance à transmettre commandes et informations beaucoup plus vite, avec un minimum de délai d'attente lors de chaque opération.

Si votre système client/serveur accède toujours au même type de source de données, et que vous êtes certain de ne jamais avoir d'autres besoins, vous pouvez utiliser le pilote natif fourni avec votre SGBD. Par contre, si vous pensez que vous devrez un jour accéder à des sources de données d'un type différent, utilisez une API ODBC pour éviter de devoir recommencer plus tard votre travail.

Les pilotes ODBC sont également optimisés pour travailler avec des sources de données spécifiques, mais ils présentent tous la même interface au gestionnaire de pilote. Un pilote qui n'a pas été optimisé pour un front end particulier n'est probablement pas aussi rapide qu'un *pilote natif* spécialement conçu pour ce front end. L'un des inconvénients des pilotes ODBC de première génération était qu'ils étaient peu performants comparés aux pilotes natifs. Toutefois, des tests récents ont démontré que les pilotes ODBC 4.0 sont devenus bien plus compétitifs. La technologie ODBC est à ce point mature qu'il n'est plus nécessaire de sacrifier la standardisation sur l'autel des performances.

ODBC et Internet

Les opérations sur une base de données sont très différentes sur Internet et sur un système client/serveur. Du point de vue de l'utilisateur, la différence la plus visible réside dans la partie client du système (qui comprend l'interface utilisateur). Dans le cas d'un système client/serveur, cette interface est la partie d'une application qui communique avec la source de données hébergée par le serveur via des instructions SQL compatibles ODBC. Sur le World Wide Web, la partie client du système est un navigateur Web qui communique avec la source de données hébergée sur le serveur via HTML (HyperText Markup Language, langage de balises hypertexte).

De nos jours, HTML n'est plus le seul langage du Web. Mais cela ne change en rien notre propos.

Toute personne pourvue d'un navigateur Web peut accéder à des données disponibles sur le Web. Le fait de mettre en ligne une base de données (on dit aussi publier) offre de nombreux avantages, surtout si vous voulez partager des informations en dehors de votre réseau local. Malheureusement, vous ne disposez généralement pas d'un contrôle très strict sur la personnalité des internautes… C'est pourquoi mettre en ligne des bases de données revient plus à publier des informations – sous-entendu pour le monde entier – qu'à les partager – sous-entendu avec quelques collègues de travail. Pour autant, ce n'est pas parce que votre navigateur est capable d'accéder à des données sur le Web qu'il sait quoi en faire. Pour plus de précisions sur ce point, voyez un peu plus loin la section « Les extensions client ». La Figure 17.1 illustre la comparaison entre systèmes client/serveur et systèmes s'appuyant sur le Web.

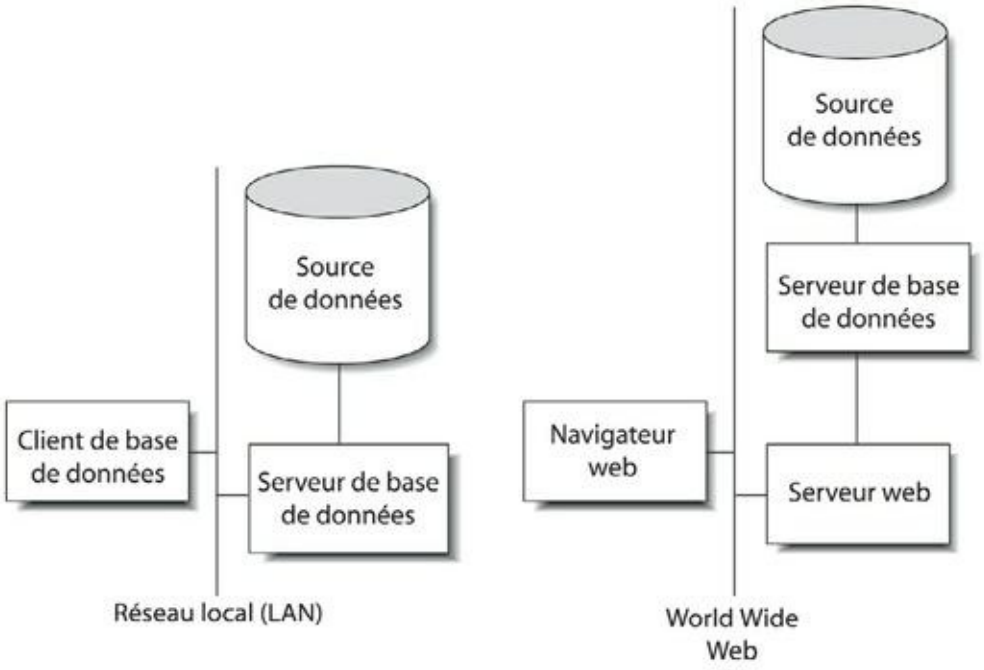


FIGURE 17.1 Système client/serveur versus système de base de données sur le Web.

Les extensions serveur

Dans un système Web, la communication entre le navigateur qui se trouve sur le poste client et le serveur Web qui se trouve sur une machine distante

s'effectue en HTML. Un composant système appelé *extension serveur* traduit le HTML en SQL compatible ODBC. Le serveur de base de données agit sur ce code SQL, qui attaque à son tour directement la source de données. Dans la direction inverse, la source de données renvoie le résultat généré par une requête à l'extension serveur via le serveur de base de données. L'extension se charge de transformer ce résultat dans une forme que le serveur Web est capable de gérer. Les réponses sont alors envoyées via le Web au navigateur qui se trouve sur la machine client. Enfin, le navigateur affiche ces résultats sur l'écran de l'utilisateur. La Figure 17.2 illustre l'architecture de ce type de système.

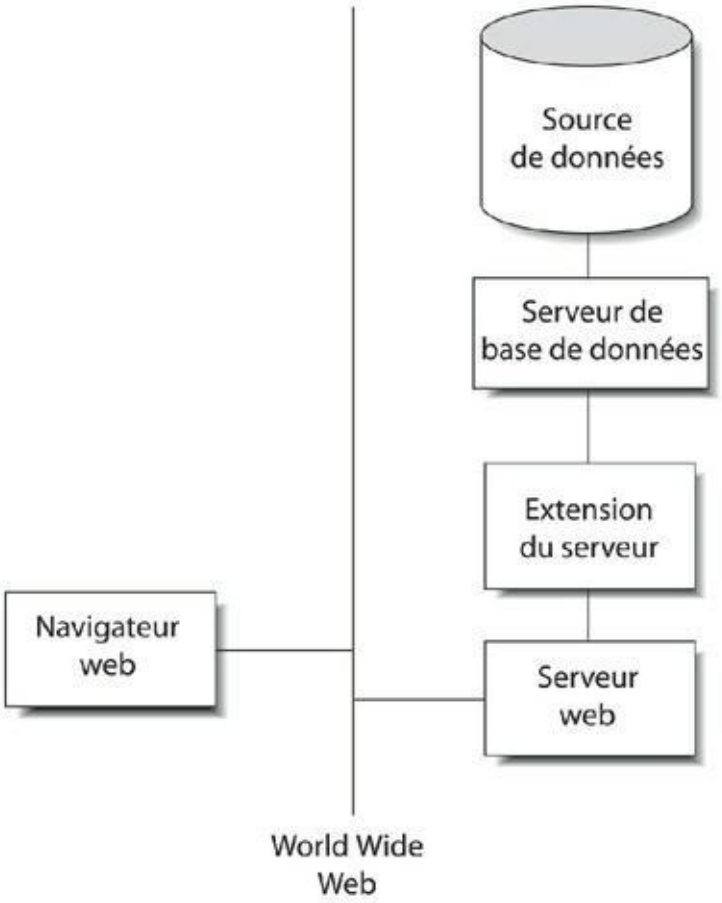


FIGURE 17.2 Système d'accès à une base de données sur le Web dotée d'une extension serveur.

Les extensions client

Certes, les navigateurs Web sont conçus et optimisés pour être faciles d'emploi et pour se connecter à toutes sortes de sites Web. Mais les

navigateurs les plus répandus (Microsoft Internet Explorer, Mozilla Firefox ou encore Apple Safari) n'ont été ni conçus ni optimisés pour servir d'interface à des bases de données. Pour manipuler une base de données via Internet, le client a besoin d'une fonctionnalité que le navigateur ne fournit pas. Pour pallier ce problème, différents types d'e*xtensions client* ont été développés. Ces extensions sont soit des applications dites d'assistance (*helper*), soit des contrôles ActiveX, soit des applettes Java, soit encore des scripts. Généralement, les extensions communiquent avec le serveur via HTML, le principal langage du Web. Un code HTML qui a besoin d'accéder à une base de données est traduit par l'extension serveur en SQL compatible ODBC avant d'être transmis à la source de données.

Les contrôles ActiveX

Les contrôles ActiveX de Microsoft fonctionnent avec Internet Explorer, de très loin le plus répandu des navigateurs Web (quoique Chrome et Mozilla Firefox commencent à lui faire un peu d'ombre).

Les scripts

Les scripts sont les outils les plus souples pour créer des extensions client. L'emploi d'un langage de scripts, comme le polyvalent JavaScript ou VBScript de Microsoft, vous permet de contrôler très précisément ce qui se passe côté client. Vous pouvez mettre en place des validations des champs de saisie, ce qui permet de rejeter ou corriger des données erronées sans avoir à les transmettre sur le Web. Cela peut vous faire gagner du temps tout en diminuant le trafic sur le Web, ce dont les autres utilisateurs vous remercieront. À l'instar des applettes Java, les scripts sont intégrés dans des pages HTML et s'exécutent lorsque l'utilisateur interagit avec ces pages.

ODBC sur un intranet

Un *intranet* est un réseau local ou étendu qui fonctionne exactement comme une version simplifiée d'Internet. Comme un intranet se limite à une organisation, vous n'avez pas besoin de mettre en place des systèmes de sécurité élaborés tels que des pare-feu. Tous les outils conçus pour le développement d'applications sur le Web permettent également de créer des applications pour un intranet.

ODBC fonctionne de la même manière sur un intranet que sur Internet. Si vous gérez plusieurs sources de données, les clients qui utilisent des navigateurs Web ainsi que les extensions appropriées peuvent communiquer avec ces sources via du code SQL qui traverse les étages HTML et ODBC de la fusée. Au niveau du pilote, le SQL compatible ODBC est traduit dans le langage de commandes natif de la base de données, puis exécuté.

JDBC

JDBC (Java DataBase Connectivity, connectivité aux bases de données pour Java) ressemble beaucoup à ODBC, mais il en diffère sur certains points importants. Comme ODBC, JDBC est une interface de bases de données qui se présente toujours de la même manière à un programme client, et ce quelle que soit la source avec laquelle il s'interface. Comme le laisse entendre son nom, une des divergences de fond avec ODBC est que JDBC fonctionne uniquement avec des applications écrites en Java et non en C++ ou en Visual Basic. Une autre différence est que JDBC et Java ont tous deux été conçus dès le départ pour fonctionner sur le Web ou sur un intranet.

Java est un langage qui ressemble au C++. Il a été conçu par Sun Microsystems dans le but spécifique de développer des programmes client pour le Web. Une fois la connexion établie entre un serveur et un client via le Web, l'applette Java appropriée est téléchargée sur le poste client, à partir de quoi elle commence à s'exécuter. L'applette, qui est intégrée dans une page HTML, fournit au client les fonctionnalités dont il a besoin pour accéder aux données hébergées sur le serveur. La Figure 17.3 est une représentation schématique d'une application de base de données pour le Web utilisant une applette Java exécutée sur la machine client.

Une *applette* est une petite application qui réside sur un serveur. Lorsqu'un client se connecte à ce serveur via le Web, l'applette est téléchargée et commence son exécution sur l'ordinateur client. Les applettes Java sont spécifiquement conçues pour fonctionner dans un *bac à sable*. Un bac à sable est une zone de mémoire parfaitement délimitée sur l'ordinateur client et dans laquelle l'applette peut s'exécuter. Elle n'est pas autorisée à déborder de son bac à sable personnel, ce qui permet de protéger la machine client d'applettes hostiles qui tenteraient de lui soustraire des informations sensibles ou de l'endommager.

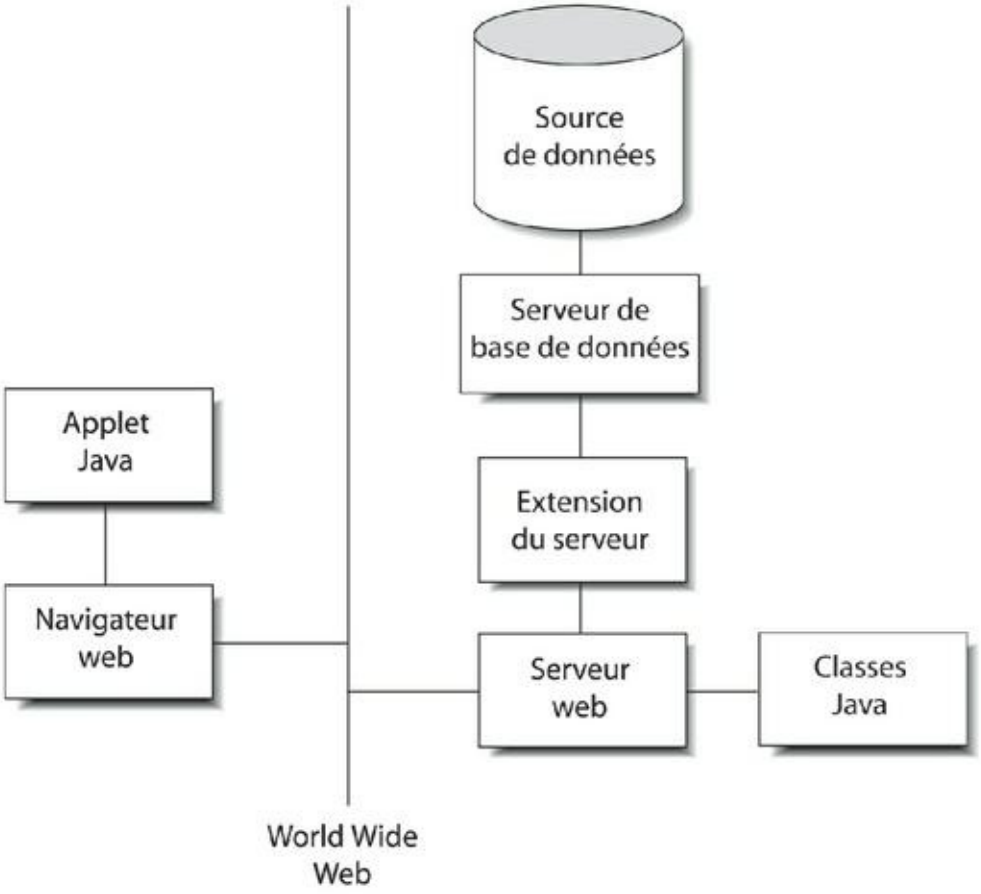


FIGURE 17.3 Application de base de données pour le Web utilisant une applette Java.

Un des principaux avantages apportés par l'utilisation des applettes Java est qu'elles sont systématiquement à jour. En effet, elles sont téléchargées depuis le serveur lors de chaque utilisation (et non conservées sur le poste du client), si bien que le client dispose toujours de la dernière version d'une applette.

Si vous êtes responsable de la maintenance d'un serveur, vous n'avez jamais besoin de vous soucier des questions de compatibilité avec vos clients lorsque vous mettez à jour le logiciel du serveur. Il vous suffit de vérifier que l'applette Java téléchargeable est compatible avec la nouvelle configuration du serveur. Si leurs navigateurs Web sont correctement configurés pour exécuter des applettes Java, tous vos clients resteront automatiquement et parfaitement compatibles. Java est un langage de programmation complet qui permet d'écrire des applications robustes servant à accéder à des bases de données sur toutes sortes de systèmes client/serveur. Ainsi utilisée, une application Java qui accède à une base de données via JDBC est semblable à une application C++ qui accède à une base de données via ODBC. Par contre, les différences de fonctionnement

apparaissent dès lors qu'il est uniquement question d'Internet (ou d'un intranet).

Si le système auquel vous comptez accéder se trouve sur Internet, les conditions de travail ne sont plus du tout les mêmes que sur un système client/serveur. La partie client d'une application qui fonctionne sur Internet est un navigateur dont les capacités de traitement sont très réduites. Ces capacités doivent être étendues pour pouvoir accéder à une base de données. C'est ce que permettent les applettes Java.

Il est toujours risqué de télécharger quoi que ce soit sur un serveur qui n'est pas totalement digne de confiance. Le risque est minime dans le cas d'une applette Java, mais il n'est pas complètement éliminé. Faites très attention avant d'installer du code exécutable sur votre machine lorsqu'il provient d'un serveur potentiellement suspect.

Comme ODBC, JDBC transmet à la source de données qui se trouve sur le back end les instructions SQL provenant de l'application front end (l'applette). Il transmet également les résultats et les messages d'erreur de la source de données vers l'application. Tout l'intérêt d'utiliser JDBC est que le développeur d'une applette peut écrire son programme en s'appuyant sur l'interface standard JDBC, sans avoir à tenir compte de la nature de la base de données située sur le back end. JDBC effectue toutes les conversions requises pour assurer une communication bidirectionnelle entre le client et la source de données.

Chapitre 18
SQL et données XML

DANS CE CHAPITRE :

- **Utiliser SQL avec XML.**
- **XML, bases de données et l'Internet.**

À partir de SQL:2008, le standard SQL de l'ISO/IEC supporte XML. Les fichiers XML (eXtensible Markup Language) sont rapidement devenus un standard universellement accepté pour l'échange de données entre plates-formes différentes. Avec XML, il n'est plus important de savoir si la personne avec laquelle vous échangez des données a ou non un environnement différent, un système d'exploitation différent ou même un matériel différent. XML est le pont qui vous permet d'échanger des données.

Liens entre SQL et XML

Comme HTML, XML est un langage de balises. Autrement dit, ce n'est pas un langage complet, à l'instar du C++ ou de Java. Il ne s'agit même pas d'un sous-langage, comme SQL. Par contre, à l'inverse des autres, il « comprend » le contenu des données qu'il transporte. HTML ne s'occupe que du formatage du texte et des graphismes. XML ne s'en soucie pas. Pour gérer la mise en forme d'un document, il faut lui associer une *feuille de style* (HTML utilisant d'ailleurs lui aussi cette méthode).

La structure d'un document XML est décrite par son schéma XML, exemple de *métadonnées* (données qui décrivent des données). Un schéma XML décrit où les éléments se trouvent dans un document et dans quel ordre ils peuvent apparaître. Il peut aussi décrire le type de données d'un élément et contraindre les valeurs que ce type peut recouvrir.

SQL et XML fournissent deux manières différentes permettant de structurer des données afin de les enregistrer et d'y retrouver les informations voulues :

- SQL est un excellent outil pour gérer des données numériques et textuelles pouvant être groupées par type et possédant une longueur définie.

  SQL a été créé dans le but de standardiser le traitement et la gestion d'informations présentes dans une base de données relationnelle.

- XML est meilleur pour travailler sur des données dont la forme n'est pas prédéfinie et qui ne peuvent être facilement rangées dans des catégories précises.

  Ce qui a motivé la création de XML, c'est essentiellement le besoin de posséder un standard universel pour le transfert de données entre plates-formes différentes et pour l'affichage sur le World Wide Web.

Les forces et les faiblesses de SQL et de XML sont complémentaires. Chacun règne en maître sur son propre domaine et forme des alliances avec l'autre afin de donner aux utilisateurs les informations qui les intéressent, quand ils le veulent et là où ils le veulent.

Le type de donnée XML

Le type XML a été introduit dans SQL:2003. Cela signifie que les implémentations conformes au standard peuvent directement enregistrer et traiter des données au format XML, sans qu'il soit besoin de les convertir au préalable dans l'un des types de SQL.

Le type XML, y compris ses sous-types, fonctionne comme un type de donnée défini par l'utilisateur (tout en étant intrinsèque à chaque implémentation). Les sous-types sont :

- `XML(DOCUMENT(UNTYPED))`

- `XML(DOCUMENT(ANY))`
- `XML(DOCUMENT(XMLSCHEMA))`
- `XML(CONTENT(UNTYPED))`
- `XML(CONTENT(ANY))`
- `XML(CONTENT(XMLSCHEMA))`
- `XML(SEQUENCE)`

Le type XML met SQL et XML en contact étroit, car il permet aux applications d'effectuer des opérations SQL sur du contenu XML, et à l'inverse des opérations XML sur du contenu SQL. Vous pouvez inclure une colonne du type XML au milieu de colonnes possédant l'un des types décrits dans le Chapitre 2 dans une jointure. Avec une base de données véritablement relationnelle, votre SGBD déterminera la manière optimale d'exécuter la requête.

Quand utiliser le type XML

Décider si vous devriez ou non enregistrer des données au format XML dépend de l'usage que vous comptez en faire. Voici quelques circonstances dans lesquelles cette technique peut être intéressante :

- Lorsque vous voulez enregistrer un bloc entier de données, puis le retrouver plus tard.
- Lorsque vous voulez pouvoir interroger l'ensemble du document XML. Certaines implémentations ont étendu la portée de l'opérateur EXTRACT pour permettre l'extraction du contenu désiré dans un document XML.
- Lorsque vous avez besoin d'un typage fort de données dans des instructions SQL. En utilisant le type XML, vous êtes certain que vous disposez de valeurs valides, et pas simplement de chaînes de caractères arbitraires.

- Lorsque vous voulez assurer la compatibilité avec de futurs (et donc pas encore définis) systèmes de stockage qui pourraient ne pas supporter des types actuels, tels que CHARACTER LARGE OBJECT, ou CLOB (voyez le Chapitre 2 pour plus d'informations sur cet objet).
- Si vous voulez profiter de futures optimisations qui ne supporteront plus que le type XML.

Voici un exemple montrant comment vous pourriez utiliser le type XML :

```
CREATE TABLE CLIENTS (
NOM_CLIENT CHARACTER (30) NOT NULL,
ADRESSE_1 CHARACTER (30),
ADRESSE_2 CHARACTER (30),
VILLE CHARACTER (25),
ETAT CHARACTER (2),
CODE_POSTAL CHARACTER (10),
TELEPHONE CHARACTER (13),
FAX CHARACTER (13),
CONTACT CHARACTER (30),
Commentaires XML(SEQUENCE) ) ;
```

L'instruction SQL qui suit va enregistrer un document XML dans la colonne `Commentaires` de la table CLIENT. Le document résultant pourrait ressembler à ceci :

```
<Comments>S
<Comment>
          <CommentNo>1</CommentNo>
          <MessageText>Est-ce que VetLab est
équipé pour
analyser
          du sang de pingouin ?</MessageText>
```

```
<ResponseRequested>Oui</ResponseRequested>
</Comment>
<Comment>
          <CommentNo>2</CommentNo>
          <MessageText>Merci pour la célérité de
l'envoi
des résultats
          Concernant le prélèvement de phoque
gris</Mes-
sageText>

<ResponseRequested>Non</ResponseRequested>
</Comment>
</Comments>
```

Quand ne pas utiliser le type XML

Ce n'est pas parce que SQL:2003 vous permet d'utiliser le type XML que vous devez toujours le faire. En fait, cela n'a même aucun sens dans de nombreuses circonstances. La plupart des données des bases relationnelles actuelles ont un format qui leur convient parfaitement. Voici quelques exemples de cas dans lesquels le type XML n'a aucune utilité :

- Lorsque les données se décomposent naturellement en une structure relationnelle, avec des tables, des lignes et des colonnes.
- Lorsque vous avez besoin de mettre à jour des parties d'un document et non pas de traiter globalement ce document.

Associer SQL et XML, XML et SQL

Pour échanger des données entre des bases SQL et des documents XML, il faut que les divers éléments de la base puissent être transcrits en éléments équivalents du document XML. Et réciproquement. Dans les sections qui suivent, j'explique quels éléments ont besoin d'être convertis.

Jeux de caractères

En SQL, le support des jeux de caractères dépend de votre implémentation. Cela signifie que DB2 (IBM) peut supporter des jeux de caractères que Microsoft SQL Server n'accepte pas. De même, SQL Server peut supporter des jeux de caractères qui ne le sont pas par Oracle. Certes, les jeux de caractères les plus courants sont universellement reconnus. Mais, dans le cas de besoins spécifiques, migrer une base de données et ses applications d'un SGBD à un autre peut être difficile.

XML ne connaît pas ces problèmes de compatibilité. Il ne se sert en effet que d'un seul jeu de caractères : Unicode. Du point de vue de l'échange de données entre XML et une implémentation SQL quelconque, c'est une bonne nouvelle. Tous les vendeurs de SGBD doivent définir une correspondance bidirectionnelle entre les chaînes de chacun de leurs jeux de caractères et Unicode. Le travail s'arrête là, précisément puisque XML n'accepte aucun autre jeu de caractères. Sinon, le problème deviendrait beaucoup plus complexe.

Identificateurs

XML est plus strict que SQL quant aux caractères autorisés dans les identificateurs. Les signes légaux dans SQL, mais illégaux pour XML, doivent être transformés en quelque chose d'acceptable avant de pouvoir être utilisés dans un document XML. SQL supporte des identificateurs délimités. Cela signifie que toutes sortes de caractères bizarroïdes, comme %, $ et &, sont légaux dès lors qu'ils sont placés entre des guillemets. Du point de vue de XML, de tels signes sont illégaux. De plus, les noms XML qui commencent par XML (quelle que soit la combinaison de majuscules et de minuscules) sont des mots réservés, et donc ne peuvent pas être employés en toute impunité.

Si, par le plus grand des hasards, vous avez utilisé ces lettres au début d'identificateurs SQL, vous devrez changer leurs noms.

Certaines règles permettent de créer un pont entre SQL et XML. En passant à XML, tous les identificateurs SQL sont convertis en Unicode. À partir de là, ceux qui sont aussi légaux pour XML demeurent inchangés. Les autres sont remplacés par un code hexadécimal prenant pour forme « `_xNNNN_` » ou « `xNNNNNNNN_` », où `N` représente un chiffre hexadécimal en majuscule. Par exemple, le trait de soulignement (_) sera représenté par « _x005F_ ». De même, « `_x003A_` » correspond à la virgule. Les valeurs hexadécimales précédentes sont précisément les codes du trait de soulignement et de la virgule en Unicode. Le cas d'identificateurs SQL débutant par les caractères *x*, *m* et *l* est réglé en préfixant ce genre d'instance par un code de la forme « `xFFFF` ».

Les conversions sont plus simples dans l'autre sens (de XML vers SQL). Tout ce qu'il y a à faire est de scanner les caractères d'un nom XML pour y chercher la présence d'une séquence « `xNNNN_` » ou « `_xNNNNNNNN` ». Il suffit alors de la remplacer par le caractère Unicode correspondant. Si un nom XML débute par la séquence « `xFFFF_` », elle doit simplement être ignorée.

En suivant ces quelques règles, vous pouvez convertir un identificateur SQL en un nom XML, puis revenir à un identificateur SQL. Par contre, la situation est moins confortable quand il s'agit de convertir un nom XML en identificateur SQL et retour.

Types de données

Le standard SQL spécifie qu'un type de donnée SQL doit être converti dans le type XML Schema le plus proche possible. L'expression « le plus proche possible » signifie que toutes les valeurs autorisées par le type SQL le seront aussi par le type XML Schema, et qu'un minimum de valeurs non permises par le type SQL sera autorisé par le type XML Schema. Dans certains cas, il est possible de réduire l'étendue des valeurs XML à la plage autorisée par le type SQL correspondant. Prenons par exemple une valeur SQL de type INTEGER. Elle appartient à l'intervalle fermé -2157483648, 2157483647. Sous XML, la valeur `maxInclusive` peut être définie comme étant égale à 2157483647, et la valeur `minInclusive` comme étant égale à - 2157483648. Voici un exemple mettant en œuvre une telle correspondance :

```
<xsd:simpleType>
<xsd:restriction base=»xsd:integer>
<xsd:maxInclusive valeur=»2157483647» />
<xsd:minInclusive valeur=»-2157483648» />
<xsd:annotation>
          <sqlxml:sqltype name=»INTEGER» />
</xsd:annotation>
</xsd:restriction>
</xsd:simpleType>
```

La section `annotation` contient des informations sur la définition du type SQL. Elle n'est pas utilisée par XML, mais elle peut être intéressante si le document est reconverti plus tard vers SQL.

Tables

Vous pouvez associer une table à un document XML. Cela vaut également pour toutes les tables d'un schéma ou d'un catalogue. Cette opération conserve les privilèges. Une personne possédant un privilège `SELECT` restreint à certaines colonnes d'une table ne pourra associer que ces colonnes au document XML. La procédure produit en réalité deux documents : l'un contient les données de la table, l'autre le schéma XML qui décrit le premier document. Voici par exemple ce que peut donner ce genre d'opération :

```
<CLIENTS>
<row>
          <PRENOM>Abe</PRENOM>
          <NOM>Abelson</NOM>
          <VILLE>Springfield</VILLE>
          <CODE>714</CODE>
          <TELEPHONE>555-1111</TELEPHONE >
</row>
<row>
          <PRENOM>Bill</PRENOM>
```

```
            <NOM>Bailey</NOM>
            <VILLE>Decatur</VILLE>
            <CODE>714</CODE>
            <TELEPHONE>555-2222</TELEPHONE >
</row>
.
.
.
</CLIENTS>
```

L'élément racine du document a pris le nom de la table. Chaque ligne de celle-ci est contenue dans un élément `<row>`. Cet élément décrit une suite de colonnes provenant de la table et portant le même nom que dans celle-ci. Enfin, chaque élément de colonne contient une valeur de donnée.

Gérer les valeurs nulles

Les données SQL peuvent être nulles (vides, non définies ou encore indéterminées). Vous devez décider de la façon de les représenter dans un document XML. Vous disposez pour cela de deux options : nil ou absent. Si vous choisissez la méthode nil, l'attribut `xsi : nil= » true »` marque les éléments de colonnes dont la valeur dans la table est nulle. Il peut être utilisé de la manière suivante :

```
<row>
<PRENOM>Abe</PRENOM>
<NOM>Abelson</NOM>
<VILLE xsi:nil=»true» />
<CODE>714</CODE>
<TELEPHONE>555-1111</TELEPHONE >
</row>
```

L'option `absent` pourrait être implémentée ainsi :

```
<row>
```

```
<PRENOM>Abe</PRENOM>
<NOM>Abelson</NOM>
<CODE>714</CODE>
<TELEPHONE>555-1111</TELEPHONE >
</row>
```

Dans ce cas, la ligne contenant la valeur nulle est tout simplement omise.

Générer le schéma XML

Dans une association entre SQL et XML, le premier document généré est celui qui contient les données. Le second concerne les informations de schéma. À titre d'exemple, considérons le schéma pour le document CLIENTS utilisé plus haut dans la section « Tables » :

```
<xsd:schema>
<xsd:simpleType name=»CHAR_15»>
          <xsd:restriction base=»xsd:string»>
                    <xsd:length value = «15» />
          </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name=»CHAR_25»>
          <xsd:restriction base=»xsd:string»>
                    <xsd:length value = «25» />
          </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name=»CHAR_3»>
          <xsd:restriction base=»xsd:string»>
                    <xsd:length value = «3» />
          </xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name=»CHAR_8»>
          <xsd:restriction base=»xsd:string»>
                    <xsd:length value = «8» />
```

```
            </xsd:restriction>
</xsd:simpleType>
<xsd:sequence>
            <xsd:element name=»PRENOM»
type=»CHAR_15» />
            <xsd:element name=»NOM» type=»CHAR_25»
/>
            <xsd:element name=»VILLE»
type=»CHAR_25» nil-
lable=»true» />
            <xsd:element name=»CODE» type=»CHAR_3»
nil-
lable=»true» />
          <xsd:element name=»TELEPHONE»
type=»CHAR_8»
nillable=»true» />

</xsd:sequence>
</xsd:schema>
```

Ce schéma est approprié si l'approche nil est utilisée pour gérer les valeurs SQL nulles. L'approche `absent` impose une définition des éléments légèrement différente. Par exemple :

```
<xsd:element name=»VILLE» type=»CHAR_25»
minOccurs=»0» />
```

Fonctions SQL opérant sur des données XML

Le standard SQL définit une série d'opérateurs, fonctions et pseudo- fonctions qui produisent un résultat XML quand on les applique à une base de données SQL, ou qui à l'inverse produisent un résultat au format SQL standard lorsqu'on les applique à des données XML. Dans les

sections qui suivent, je donne une brève description des fonctions les plus courantes que sont XMLELEMENT, XMLFOREST, XMLCONCAT et XMLAGG, ainsi que de plusieurs autres qui sont fréquemment employées pour publier sur le Web. Certaines d'entre elles reposent fortement sur XQuery, un nouveau langage de requête standard spécifiquement conçu pour interroger des données XML. XQuery est en lui-même un vaste sujet, qui dépasse évidemment le cadre de ce livre.

XMLDOCUMENT

L'opérateur `XMLDOCUMENT` prend une valeur XML en entrée et retourne une autre valeur XML en sortie. La nouvelle valeur XML est un nœud document construit selon les règles du constructeur d'un document dans XQuery.

XMLELEMENT

L'opérateur `XMLELEMENT` traduit une valeur relationnelle en un élément XML. Vous pouvez l'utiliser dans une instruction `SELECT` pour mettre des données provenant de la base SQL au format XML en vue de les publier sur le Web. Par exemple :

```
SELECT c.NOM
XMLELEMENT (NAME «VILLE», c.VILLE) AS «Resultat»
FROM CLIENTS c
WHERE NOM=»Abelson»  ;
```

Voici le résultat renvoyé :

Nom	Résultat
Abelson	`<VILLE>Springfield</VILLE>`

XMLFOREST

L'opérateur XMLFOREST produit une liste, ou forêt, d'éléments XML à partir de valeurs relationnelles. Chacune des valeurs de l'opérateur génère un nouvel élément. En voici un exemple :

```
SELECT c.NOM
XMLFOREST (c.VILLE,
           c.CODE,
           c.TELEPHONE) AS «Resultat»
FROM CLIENTS c
WHERE NOM = «Abelson» OR NOM = «Bailey»  ;
```

Ce fragment de code donnera la sortie suivante :

Nom	Résultat
Abelson	<VILLE>Springfield</VILLE> <CODE>714</CODE> <TELEPHONE>555-1111</TELEPHONE>
Bailey	<VILLE>Decatur</VILLE> <CODE>714</CODE> <TELEPHONE>555-2222</TELEPHONE>

XMLCONCAT

XMLCONCAT représente une autre manière de produire une forêt d'éléments en concaténant ses arguments XML. Par exemple, le code ci-dessous :

```
SELECT c.NOM
XMLCONCAT (
           XMLELEMENT ( NOM «prenom», c.PRENOM)
           XMLELEMENT ( NOM «nom», c.NOM)
) AS «Resultat»
FROM CLIENTS c ;
```

produit ces résultats :

Nom	Résultat
Abelson	<prenom>Abe</prenom> <nom>Abelson</nom>
Bailey	<prenom>Bill</prenom> <nom>Bailey</nom>

XMLAGG

La fonction d'agrégation XMLAGG prend des documents XML (ou des fragments de documents) et produit un nouveau document unique comme sortie d'une requête GROUP BY. L'agrégation contient une forêt d'éléments. Voyons un exemple qui illustre ce concept :

```
SELECT XMLELEMENT
( NAME «Ville»
          XMLATTRIBUTES ( c.VILLE AS «name» )
          XMLAGG ( XMLELEMENT (NAME «nom» c.NOM
))
) AS «ListeVilles»
FROM CLIENTS c
GROUP BY VILLE ;
```

Lorsqu'elle est exécutée sur notre table CLIENTS, cette requête fournit le résultat suivant :

```
ListeVilles
<Ville name=»Decatur»>
<nom>Bailey</nom>
</Ville>
<Ville name=»Philo»>
<nom>Stetson</nom>
<nom>Stetson</nom>
<nom>Wood</nom>
</Ville>
```

```
<Ville name=»Springfield»>
<nom>Abelson</nom>
</Ville>
```

XMLCOMMENT

La fonction `XMLCOMMENT` permet à une application de créer un commentaire XML. Sa syntaxe est la suivante :

```
SXMLCOMMENT ( ‘contenu du commentaire’
 [RETURNING
          {CONTENT | SEQUENCE} ] )
```

Par exemple :

```
XMLCOMMENT (‘Sauvegarder la base tous les jours à
2h du
matin.)
```

crée un commentaire XML semblable à celui-ci :

```
<!--Sauvegarder la base tous les jours à 2h du
matin. -->
```

XMLPARSE

La fonction `XMLPARSE` produit une valeur XML en effectuant une analyse sans validation d’une chaîne. Vous pourriez l’utiliser de cette manière :

```
XMLPARSE (DOCUMENT ‘ Beau travail ! ‘
PRESERVE WHITESPACE )
```

Le code ci-dessus renverrait une valeur XML du type `XML (UNTYPED DOCUMENT)` ou `XML (DOCUMENT)`. Le sous-type choisi dépend de l’implémentation que vous utilisez.

XMLPI

La fonction `XMLPI` permet aux applications de créer des instructions de traitement XML. Sa syntaxe se présente ainsi :

```
XMLPI NAME cible
 [, expression-chaîne ]
 [ RETURNING
          { CONTENT | SEQUENCE } ] )
```

Le container `cible` désigne l'identificateur de la cible visée par l'instruction de traitement. `expression-chaîne` représente le contenu de cette instruction. Cette fonction crée un commentaire XML de la forme :

```
<? cible expression-chaîne ?>
```

XMLQUERY

La fonction `XMLQUERY` évalue une expression XQuery et renvoie le résultat de la requête à l'application SQL. La syntaxe de `XMLQUERY` est la suivante :

```
XMLQUERY ( expression-XQuery
[ PASSING { BY REF | BY VALUE }
liste-arguments ]
            [ RETURNING { CONTENT | SEQUENCE }
{ BY REF | BY VALUE } )
```

Voici un exemple d'utilisation de XMLQUERY :

```
SELECT moyenne_max,
XMLQUERY (
          'for $moyenne_batteur in
                    /joueur/moyenne_batteur
          where /joueur/nom = $var1
```

```
            return $moyenne_batteur'
            PASSING BY VALUE
                     'Mantle' AS var1
            RETURNING SEQUENCE BY VALUE )
FROM stats_attaque ;
```

XMLCAST

La fonction `XMLCAST` est semblable au CAST de SQL, mais avec quelques restrictions supplémentaires. Elle permet à une application de pratiquer le *transtypage* d'une valeur d'un type XML vers un autre type XML ou vers un type SQL. Elle peut aussi servir dans l'autre sens, pour transformer un type SQL en un type XML. Mais il y a quelques contraintes à respecter :

- Au moins un des types invoqués (la source ou la destination) doit être un type XML.
- Les types SQL invoqués ne peuvent être ni une collection, ni une ligne, ni un type structuré, ni une référence.
- Seules les valeurs possédant l'un des types XML ou le type SQL nul peuvent être transformées en XML (UNTYPED DOCUMENT) ou XML (DOCUMENT).

Voici un exemple :

```
XMLCAST ( CLIENTS.NomClient AS XML (UNTYPED
DOCUMENT)
```

La fonction `XMLCAST` est transformée en un CAST SQL ordinaire. La seule raison qui justifie l'emploi d'un mot clé différent est la nécessité de respecter les contraintes ci-dessus.

Prédicats

Un *prédicat* renvoie une valeur vraie (True) ou fausse (False). De nouveaux prédicats ont été ajoutés pour les besoins de l'interconnexion entre SQL et XML.

DOCUMENT

L'objet du prédicat `DOCUMENT` est de déterminer si une valeur XML est un document. Il teste cette valeur pour vérifier si elle est une instance du type `XML (UNTYPED DOCUMENT)` ou `XML (ALL DOCUMENT)`. Sa syntaxe est la suivante :

```
XML-valeur IS [NOT]
  [ANY | UNTYPED] DOCUMENT
```

Si l'expression est évaluée comme étant vraie, le prédicat retourne True. Sinon, il renvoie False. Si la valeur XML est nulle, le prédicat retourne UNKNOWN. Si `ANY` ou `UNTYPED` ne sont pas spécifiés, le prédicat suppose par défaut qu'il s'agit d'`ANY`.

CONTENT

Le prédicat `CONTENT` sert à déterminer si une valeur XML est une instance de `XML (ANY CONTENT)` ou `XML (UNTYPED CONTENT)`. En voici la syntaxe :

```
XML-valeur IS [NOT]
  [ANY | UNTYPED] CONTENT
```

Si `ANY` ou `UNTYPED` ne sont pas spécifiés, le prédicat suppose par défaut qu'il s'agit d'`ANY`.

XMLEXISTS

Comme son nom l'indique, le prédicat `XMLEXISTS` a pour but de déterminer si une valeur existe. Sa syntaxe se présente ainsi :

```
XMLEXISTS (expression-XQuery
 [ liste-arguments ] )
```

L'expression XQuery utilise les valeurs fournies dans la liste des arguments. Si cette expression renvoie une valeur SQL nulle, le résultat fourni par le prédicat est UNKNOWN. Si l'évaluation retourne une séquence XQuery vide, le prédicat vaut FALSE. Sinon, il est vrai (TRUE). Ce prédicat peut servir à déterminer si un document XML contient un contenu spécifique avant d'utiliser une partie de ce contenu dans une expression.

VALID

Le prédicat VALID a pour but d'évaluer une valeur XML afin de voir si elle est valide dans le contexte d'un schéma XML enregistré. Sa syntaxe est plus complexe que celle des autres prédicats :

```
XML-valeur IS [NOT] VALID
 [XML option de contrainte d'identité valide]
 [XML clause de correspondance valide]
```

Le prédicat contrôle si la valeur possède bien l'un des cinq types XML : XML (SEQUENCE), XML (ANY CONTENT), XML (UNTYPED CONTENT), XML (ANY DOCUMENT), XML (UNTYPED DOCUMENT). De plus, il peut vérifier de manière facultative si la validité de la valeur XML s'appuie sur des contraintes d'identité, et si cette validité est conforme à un schéma XML particulier.

Le composant option de contrainte d'identité offre quatre possibilités :

- WITHOUT IDENTITY CONSTRAINTS : C'est la supposition qui est faite par défaut si aucune autre option de contrainte d'identité n'est utilisée. Si DOCUMENT est spécifié, elle fonctionne comme une combinaison des prédicats DOCUMENT et VALID WITH IDENTITY CONSTRAINTS GLOBAL.

- `WITH IDENTITY CONSTRAINTS GLOBAL` : Ce composant demande à vérifier la conformité de la valeur avec le schéma XML ainsi qu'avec les règles XML pour les relations ID/ IDREF.

  ID et IDREF sont des types d'attributs XML qui identifient les éléments d'un document.

- `WITH IDENTITY CONSTRAINTS LOCAL` : Ce composant demande à vérifier la conformité de la valeur avec le schéma XML, mais pas avec les règles XML pour les relations ID/ IDREF ou les règles du schéma XML pour les contraintes d'identité.

- `DOCUMENT` : Ce composant signifie que la valeur est un document et qu'elle respecte le mode `WITH IDENTITY CONSTRAINTS GLOBAL` avec une clause de correspondance valide. Celle-ci identifie le schéma pour lequel la valeur doit être validée.

Transformer des données XML en tables SQL

Jusqu'à une période récente, la réflexion sur les relations entre SQL et XML tournaient essentiellement autour de la question suivante : « Comment convertir des données d'une table SQL au format XML pour les rendre accessibles sur l'Internet ? » Les plus récents ajouts au standard SQL traitent aussi du problème inverse : « Comment convertir des données XML en tables SQL pour pouvoir les interroger facilement à l'aide d'instructions SQL standard ? »

La pseudo-fonction `XMLTABLE` est chargée d'effectuer cette opération. Sa syntaxe est la suivante :

```
XMLTABLE ( [déclaration-espace-noms, ]
expression-XQuery
[ PASSING liste-arguments ]
COLUMNS définitions-colonnes-tableXML
```

où la liste d'arguments se présente ainsi :

```
expression-valeur AS identificateur
```

Tandis que `définitions-colonnes-tableXML` est une liste de définitions de colonnes séparées par une virgule et qui peut contenir :

```
nom-colonne FOR ORDINALITY
```

et/ou :

```
nom-colonne type-donnée
[BY REF | BY VALUE]
[clause-par-défaut]
[PATH expression-XQuery]
```

Voici un exemple vous montrant comment il est possible d'utiliser `XMLTABLE` pour extraire des données d'un document XML et les envoyer vers une pseudo-table SQL. Une telle table a une durée de vie éphémère, mais elle se comporte pour le reste comme une table SQL tout à fait normale. Si vous voulez que les données soient permanentes, vous pouvez faire appel à une instruction `CREATE TABLE` puis insérer les données XML dans la table nouvellement créée :

```
SELECT telephoneclient.*
FROM
clients_xml ,
XMLTABLE (
          'for $m in
                   ol/client
          return
```

```
                    $m'
PASSING clients_xml.client AS «col»
COLUMNS
          «Nom_Client» CHARACTER (30) PATH
'Nom_Client' ,
          «Telephone» CHARACTER (13) PATH
'Telephone'
) AS telephoneclient ;
```

L'exécution de cette instruction fournira un résultat de ce style :

```
Nom_Client          Telephone
Abe Abelson         (714) 555-1111
Bill Bailey         (714) 555-2222
Chuck Wood          (714) 555-3333
```

Types de données non prédéfinis et XML

Dans le standard SQL, les types de données non prédéfinis comprennent les domaines, les types utilisateurs distincts, les tableaux et les multisets (ensembles multiples). Vous pouvez tous les transformer en données formatées pour XML en utilisant le code approprié. Les quelques sections qui suivent vous proposent des exemples simples.

Domaine

Pour convertir un domaine SQL vers XML, vous devez tout d'abord disposer d'un domaine. Pour cet exemple, nous allons en créer un grâce à l'instruction `CREATE DOMAIN` :

```
CREATE DOMAIN CoteOuest AS CHAR (2)
CHECK (Etat IN ('CA', 'OR', 'WA', 'AK') ) ;
```

Définissons maintenant une table qui utilise ce domaine :

```
CREATE TABLE RegionOuest (
NomClient Character (20)    NOT NULL,
Etat CoteOuest     NOT NULL
)  ;
```

Voici le schéma XML permettant de transcrire notre domaine en XML :

```
<xsd:simpleType>
Name = ‘DOMAIN.Ventes.RegionOuest’>

<xsd:annotation>
<xsd:appinfo>
          <sqlxml:sqltype kind = ‘DOMAIN’
                    schemaName = ‘Ventes’
                    typeName = ‘CoteOuest’
                    mappedType = ‘CHAR_2’
                     final = ‘true />
          </xsd:appinfo>
</xsd:annotation>

<xsd:restriction base = ‘CHAR_2’ />

</xsd:simpleType>
```

Lorsque ce mappage est appliqué, il en résulte un document XML qui contient quelque chose comme :

```
<RegionOuest>
<row>
.
.
.
<Etat>AK</Etat>
.
.
```

```
.
</row>
.
.
.
</RegionOuest>
```

Type défini par l'utilisateur distinct

Le travail est à peu près le même que pour un domaine, ainsi que le montre l'exemple suivant :

```
CREATE TYPE CoteOuest AS CHARACTER (2) FINAL ;
```

Voici le schéma XML permettant de transcrire notre type en XML :

```
<xsd:simpleType>
Name = 'UDT.Ventes.RegionOuest'>

<xsd:annotation>
<xsd:appinfo>
          <sqlxml:sqltype kind = 'DISTINCT'
                   schemaName = 'Ventes'
                   typeName = 'CoteOuest'
                   mappedType = 'CHAR_2'
                    final = 'true' />
          </xsd:appinfo>
</xsd:annotation>

<xsd:restriction base = 'CHAR_2' />

</xsd:simpleType>
```

Il crée un élément identique à celui obtenu avec le domaine précédent.

Ligne

Le type ROW vous permet de récupérer d'un coup l'ensemble des informations d'une ligne dans un seul champ. Vous pouvez créer un type ROW dans une définition de table en procédant comme suit :

```
CREATE TABLE InfoContact (
Nom Character (30) ,
Telephone ROW (Maison CHAR (13), Travail CHAR
(13))
)  ;
```

Il est maintenant possible d'effectuer un mappage de ce type vers XML avec le schéma suivant :

```
<xsd:complexType Name='ROW.1'>

<xsd:annotation>
<xsd:appinfo>
          <sqlxml:sqltype kind = 'ROW'
                    <sqlxml:field name = 'Maison'
                              mappedType =
'CHAR_13' />
                    <sqlxml:field name =
'Travail'
                              mappedType =
'CHAR_13' />
                    </sqlxml:sqltype>
          </xsd:appinfo>
</xsd:annotation>

<xsd:sequence>
          <xsd:element Name= 'Maison' nillable=
'true'
```

```
                    Type= ‘CHAR_13’ />
          <xsd:element Name= ‘Travail’ nillable=
‘true’
                    Type= ‘CHAR_13’ />
</xsd:sequence>

</xsd:complexType>
```

Ce schéma pourrait générer la séquence XML suivante pour une colonne :

```
<Telephone>
<Maison>(888) 555-1111</Maison>
<Travail>(888) 555-1111</Travail>
</Telephone>
```

Tableau

Il est possible de placer plusieurs éléments dans un seul champ en utilisant le type `ARRAY` à la place de ROW. Reprenons l’exemple de la table `InfoContact` en déclarant la colonne `TELEPHONE` comme étant un tableau :

```
CREATE TABLE InfoContact (
Nom Character (30) ,
Telephone Character (13) ARRAY [4]
)  ;
```

Il est maintenant possible d’effectuer un mappage de ce type vers XML avec le schéma suivant :

```
<xsd:complexType Name=‘ARRAY_4.CHAR_13’>
<xsd:annotation>
          <xsd:appinfo>
                    <sqlxml:sqltype kind =
‘ARRAY’
```

```
                                    maxElements= ‘4’
                                    mappedElementType=
‘CHAR_13’
/>
          </xsd:appinfo>
</xsd:annotation>

<xsd:sequence>
          <xsd:element Name= ‘element’
          minOccurs= ‘0’ maxOccurs= ‘4’
          nillable= ‘true’ Type= ‘CHAR_13’ />
</xsd:sequence>
</xsd:complexType>
```

Ce schéma générerait quelque chose comme :

```
<Telephone>
<element>(888) 555-1111</element>
<element>xsi:nil= ‘true’ />
<element>(888) 555-3434</element>
</Telephone>
```

L’élément du tableau qui contient `xsi : nil= ‘true’` reflète le fait que le second numéro de téléphone contient une valeur nulle dans la table source.

Multiset

Les numéros de téléphone de l’exemple précédent pourraient être enregistrés dans un multiset (un ensemble multiple) à la place d’un tableau. La définition de notre table se présenterait alors ainsi :

```
CREATE TABLE InfoContact (
Nom Character (30) ,
Telephone Character (13) MULTISET
```

```
)  ;
```

Il est maintenant possible d'effectuer un mappage de ce type vers XML avec le schéma suivant :

```
<xsd:complexType Name='MULTISET.CHAR_13'>

<xsd:annotation>
          <xsd:appinfo>
                    <sqlxml:sqltype kind =
'MULTISET'
                                   mappedElementType=
'CHAR_13'
/>
          </xsd:appinfo>
</xsd:annotation>

<xsd:sequence>
          <xsd:element Name= 'element'
          minOccurs= '0' maxOccurs= 'unbounded'
          nillable= 'true' Type= 'CHAR_13' />
</xsd:sequence>

</xsd:complexType>
```

Ce schéma générerait quelque chose comme ceci :

```
<Telephone>
<element>(888) 555-1111</element>
<element>xsi:nil= 'true' />
<element>(888) 555-3434</element>
</Telephone>
```

Le mariage de SQL et de XML

SQL fournit une méthode standard universelle pour le stockage de données d'une manière hautement structurée. La structure permet à l'utilisateur de gérer des données de type et de taille très différents, ainsi que d'extraire de ces banques de données les informations dont il a besoin. XML est devenu un standard officiel pour le transport de données entre plates-formes incompatibles, en particulier sur l'Internet. En associant ces deux méthodes, leur valeur individuelle s'en trouve démultipliée.

Sommaire