Table des matières

Petit manuel SQL

Pascal Ostermann

7 mars 2010

Ceci ne se veut pas un manuel complet, mais un résumé des points importants de SQL. Le plan même du texte, basé sur la structure de ce langage, le rend impropre à se substituer au cours : certains concepts, tel le SELECT imbriqué, font apparition avant d'être définis. Afin de simplifier la recherche dans ce document, il est précédé d'une table des matières et suivi d'un index.

Quelques généralités

Une **commande** sera une formule ayant un sens dans SQL. J'appellerai souvent une telle commande par le mot-clef la débutant. Une telle commande se compose de plusieurs **clauses**, que je baptiserai également du mot-clef la débutant. Ainsi une "commande SELECT" – la commande dans son ensemble, jusqu'à exécution – comportera une "clause FROM" – tout ce qui suit le FROM, jusqu'à la clause suivante –, une "clause WHERE", etc. . ., mais aussi une "clause SELECT" : tout ce qui suit le SELECT jusqu'au FROM.

Au sujet du script SQL standard

Majuscules et minuscules sont généralement interchangeables (sauf bien sûr dans les chaînes de caractères). Les retours-chariots sont équivalents à des espaces. Le point-virgule est nécessaire pour terminer une commande SQL : point-virgule suivi du retour chariot exécute la commande courante et l'enregistre dans le buffer d'édition, alors que deux retours-chariot successifs enregistrent la commande sans exécution. Il y a bien sûr des commandes permettant de modifier et d'exécuter une commande préalablement enregistrée dans le buffer d'édition. . . mais cela n'a pas sa place dans un "petit" manuel SQL.

1 Définition des données

1.1 CREATE TABLE

```
CREATE TABLE nom_table (att1 dom1 [NULL | NOT NULL], ...)};
```

définit une relation de nom *nom_table*. Son premier attribut, de nom *att1*, est de domaine *dom1* et peut être facultatif (NULL) ou obligatoire (NOT NULL) :

```
CREATE TABLE Emp
  (num_emp NUMBER NOT NULL,
   nom CHAR(15) NOT NULL,
   salaire NUMBER(7,2) NOT NULL,
   prime NUMBER(6,2),
   embauche DATE NOT NULL,
   dept CHAR(10));
```

NUMBER(7,2) signifie 7 chiffres, dont 2 après la virgule ; et DATE est un type prédéfini comprenant date et heure. Notez que l'attribut *prime* peut prendre la valeur NULL (ainsi que *dept*, mais cela ne va pas durer...) : on peut également écrire *prime NUMBER(6,2) NULL*

```
CREATE TABLE Dept (nom_dept CHAR(10) NOT NULL,
               budget NUMBER(12,2), num_chef NUMBER);
```

1.2 ALTER TABLE

```
ALTER TABLE nom_tab [ADD... | MODIFY...];
```

permet de modifier la définition d'une table (ne pas confondre avec la commande UPDATE du LMD, qui en modifie le **contenu**).

```
ALTER TABLE Emp ADD (prenom CHAR(10));
```

ajoute l'attribut facultatif *prenom* à la définition de la relation. Un attribut obligatoire ne peut être ajouté qu'à une table vide.

```
ALTER TABLE Emp MODIFY (dept NOT NULL, nom CHAR(20));
```

L'attribut *dept* devient obligatoire, mais garde le même domaine ; l'attribut *nom* prend le type CHAR(20), mais reste obligatoire.

1.3 DESC[RIBE]

vérifie la définition d'une table ou d'une vue. Ainsi *DESC Dept* donne ici :

COLUMN	TYPE	NULL
nom_dept	CHAR(10)	NOT NULL
budget	NUMBER(12,2)	NULL
num_chef	NUMBER	NULL

1.4 DROP TABLE nom_tab ;

détruit une table (et son contenu !).

1.5 Vues

1.5.1 CREATE VIEW

Une **vue** est la matérialisation d'un calcul sous une forme similaire à une table. Elle doit donc avoir un nom, et ses colonnes prendre des noms d'attributs. Si elles n'ont pas a priori de tels noms, il faut renommer. Ainsi dans

```
CREATE VIEW Effectif AS
SELECT dept, COUNT(*) effectif FROM Emp GROUP BY dept;
```

il faut renommer le COUNT(*). Une autre méthode de renommage :

```
CREATE VIEW Chef(num_emp, nom_emp, num_chef, nom_chef) AS
  SELECT E.num_emp,E.nom,C.num_emp,C.nom
  FROM Emp E, Dept, Emp C
  WHERE E.dept=num_dept AND num_chef=C.num_emp;
```

Remarques

- La définition d'une vue ne contient ni DISTINCT ni ORDER BY.
- Une vue peut rarement être mise-à-jour, et seulement si la mise-à-jour se définit sans ambiguîté sur les tables concernées.

1.5.2 DROP VIEW nom_vue ;

permet de détruire une vue.

1.6 Contraintes

Tous les SQL ne permettent pas de définir des contraintes. Ceux qui le permettent le font

- soit par la donnée, après une "phrase" du type CREATE CONSTRAINT nom_contrainte AS, d'une condition de WHERE ;
- soit à l'intérieur d'un CREATE TABLE ou ALTER TABLE.

Ces contraintes sont nommées et associées à une relation. Les plus utiles :

- les **clefs** – certains SQL, qui n'ont pas explicitement de contraintes, permettent pourtant de définir des clefs via un UNIQUE INDEX (voir le terme INDEX) – bien entendu, si le schéma est normalisé, il n'est pas besoin d'autres dépendances que la donnée des clefs ;
- les **contraintes d'inclusion** – ainsi tout *num_chef* (numéro de chef) de *Dept* est un numéro d'employé, et existe en tant que *num_emp* de *Emp* : au besoin, cette contrainte d'inclusion sera associée à la table *Dept* ;
- les "NOT NULL" précédents sont également des contraintes et peuvent être nommés, bien que l'utilité de ces noms m'échappe.

2 Mises-à-jour

2.1 INSERT INTO

Deux usages :

2.1.1 VALUES

permet d'insérer un n-uplet :

```
INSERT INTO Emp VALUES
  (412, 'Martin', 12000.14, NULL, '29-apr-08','Info');
```

Notez les "quotes" encadrant chaînes de caractères et dates, ainsi que le format standard de ces dernières : *quantième-3 premières lettres du mois en anglais-2 derniers chiffres de l'année*. L'heure n'est pas précisée : elle est tronquée à midi.

Ou, si l'on a oublié l'ordre des attributs :

```
INSERT INTO Emp (nom, num_emp, salaire, dept, embauche)
  VALUES ('Martin', 412, 12000.14, 'Info', '29-apr-08');
```

La *prime* n'étant pas précisée, elle prend la valeur NULL.

2.1.2 SELECT

On peut aussi, plus rarement, insérer la réponse à un SELECT :

```
INSERT INTO Emp SELECT * FROM Emp_d'une_succursale;
```

2.2 DELETE FROM

```
DELETE FROM nom_table WHERE condition;
```

enlève de la table tous les n-uplets vérifiant la *condition*. La clause WHERE est analogue à celle du SELECT. Cependant, on ne peut effectuer des suppressions que sur une relation à la fois. Si la condition utilise des attributs d'une autre relation, on doit donc l'exprimer par un SELECT imbriqué. Par exemple,

```
DELETE FROM Emp WHERE dept IN
(SELECT num_dept FROM Dept WHERE budget = 0);
```

supprime tous les employés des départements dont le *budget* est égal à 0.

Attention ! Sans WHERE, DELETE supprime tout le contenu de la relation.

2.3 UPDATE

```
UPDATE nom_table SET action WHERE condition;
```

applique l'*action* à tous les n-uplets vérifiant la *condition*. Comme pour le DELETE, on ne met à jour qu'une relation...

```
UPDATE Emp SET salaire = salaire * 1.1, prime = 0
WHERE dept = 'Info';
```

augmente de 10 pour cent le salaire et met à zéro la prime de chacun des employés du département d'informatique

Attention ! Sans condition, UPDATE modifie toute la relation.

2.4 COMMIT et ROLLBACK

Lorsque vous réalisez des mises-à-jour sur une relation R, elles ne sont pas encore transmises à la base de données :

- R est bloquée en mise-à-jour pour les autres utilisateurs. Ceux-ci ont le droit de la regarder, auquel cas ils n'en verront que l'état initial – avant réalisation effective de la mise-à-jour –, mais toute mise-à-jour de leur part sera bloquée jusqu'à validation de votre mise-à-jour bloquante.
- Le résultat de votre mise-à-jour apparaît dans votre espace de travail, et vous pouvez réaliser d'autres mises-à-jour sur R ainsi modifiée.

L'instruction validant les mises-à-jour s'appelle COMMIT. La commande ROLLBACK ramène la base de données à l'état du dernier COMMIT. On peut toutefois demander un COMMIT automatique à chaque mise-à-jour, en positionnant une "variable" AUTOCOMMIT à ON.

3 Requêtes

3.1 Clauses SELECT et FROM : projection

SELECT est l'opérateur principal du langage de requêtes. Le mot-clef SELECT est immédiatement suivi des valeurs que l'on désire afficher (ou pour un SELECT imbriqué, des valeurs employées dans la commande externe). Ces valeurs, éventuellement préfixées et / ou renommées, sont :

- des noms d'attributs (voir aussi "ambiguïté") ou
- des résultats de calcul (cf. "fonctions" et "fonctions d'agrégation").

Existe aussi l'abréviation * pour "tous les attributs de la table considérée" :

```
SELECT * FROM Emp;
```

consultation de la table *Emp*.

```
SELECT dept FROM Emp;
```

liste des départements. Malheureusement, chaque département figurera autant de fois qu'il a d'employés ; pour ne les avoir qu'une fois chacun (et obtenir la **projection** sur l'attribut *dept* de la relation *Emp*) :

```
SELECT DISTINCT dept FROM Emp;
```

On peut également classer la sortie d'un SELECT via la clause ORDER BY (ne pas confondre avec GROUP BY) :

```
SELECT * FROM Emp ORDER BY embauche DESC, nom;
```

classe les employés par ordre décroissant d'embauche, puis, pour deux employés embauchés simultanément, par ordre alphabétique (croissant par défaut : une formulation équivalente est *ORDER BY embauche DESC, nom ASC*).

Clauses de présentation d'un résultat, ni DISTINCT ni ORDER BY ne figurent dans une vue ou dans un SELECT imbriqué.

Syntaxe abrégée d'un SELECT :

```
SELECT [DISTINCT] attribut1, attribut2...
FROM relation1 renommage_rel1, relation2, renomm2, etc...
[WHERE conditions de selection et de jointure, s'il y a lieu]
[GROUP BY divers_noms_d'attribut
[HAVING conditions sur les groupes definis ci-dessus]]
[ORDER BY un_ou_plusieurs_noms_d'attribut];
```

La clause FROM suit immédiatement la clause SELECT. Elle contient les noms des tables utiles au calcul de la requête (avec éventuellement des renommages). Elle ne les contient pas nécessairement toutes, mais au moins toutes celles dont un attribut apparaît dans la clause SELECT.

Si deux tables figurent dans la clause FROM, le SELECT en fait le produit cartésien. Pour faire la jointure (même naturelle), **il faut écrire explicitement la condition de jointure**, comme plus bas.

3.2 WHERE : sélection et jointure

La clause WHERE permet de réaliser une **sélection** :

```
SELECT * FROM Emp WHERE dept = 'Info';
```

Des conditions plus complexes se réalisent à l'aide des opérateurs booléens NOT (négation), AND (et logique), OR (ou logique) et des parenthèses :

```
SELECT num_emp, nom, prenom FROM Emp
WHERE (embauche > '23-feb-90' OR dept <> Info')
  AND NOT (salaire < 10000 AND nom <> 'Martin');
```

Un WHERE peut aussi exprimer une condition de **jointure** :

```
SELECT num_emp, num_chef FROM Emp, Dept WHERE dept=nom_dept;
```

Attention ! Ne pas oublier la condition de jointure : si elle manque, SQL ne donne ni message d'erreur ni *warning*, mais **un résultat faux**.

Une autre expression de la jointure est la clause JOIN, dont la variante la plus intéressante est l'OUTER JOIN :

```
SELECT num_emp, num_chef
FROM Emp LEFT [OUTER] JOIN Dept ON dept=nom_dept;
```

Si un département n'a pas d'employés, il est joint à un employé fictif dont tous les attributs sont NULL.

Il existe également RIGHT et FULL [OUTER] JOIN, [INNER] JOIN ; de plus la condition de la jointure naturelle (équijointure des attributs de même nom) s'exprime en faisant précéder chacune de ces formules par NATURAL.

3.3 Ambiguïté de noms d'attributs et renommages

Il est permis de donner un même nom à deux attributs de deux tables différentes. Si ces deux attributs sont utilisés dans une même requête, on les distingue en les préfixant du nom de la relation, suivi d'un point :

Si par exemple dans *Dept* et *Emp* le même nom *nom_dept* est donné aux attributs *nom_dept* (de *Dept*) et *dept* (de *Emp*), la jointure (naturelle) s'écrit :

```
SELECT num_emp, num_chef FROM Emp, Dept
WHERE Emp.nom_dept = Dept.nom_dept;
```

Le même préfixage permet d'utiliser une table issue d'un autre schéma :

```
SELECT num_emp, num_chef
FROM Emp NATURAL [INNER] JOIN Klein.Dept;
```

On peut avoir à utiliser deux fois la même relation dans une même requête (jointure d'une table sur elle-même). Il faut alors renommer au moins l'une de ces occurrences. L'opérateur de renommage de SQL est l'espace. Ainsi, pour obtenir nom et nom du chef de département de chaque employé :

```
SELECT E.nom, C.nom FROM Emp E, Dept, Emp C
WHERE E.dept=num_dept AND num_chef=C.num_emp;
```

La première occurrence de *Emp* est renommée *E*, l'autre *C*.

Les tables sont renommées dans une clause FROM ou apparentée. On peut également renommer des attributs (dans le SELECT – c'est utile pour définir certaines vues).

3.4 Calcul relationnel en SQL

3.4.1 Prédicats du premier ordre

Nous disposons des comparateurs classiques =, <>, <, <=, >, >=. Appliqués à des chaînes de caractères ou à des dates, ils se réfèrent à l'ordre lexicographique (qui n'est pas l'ordre alphabétique du français) ou chronologique.

Il existe également des prédicats spécifiques à divers domaines, tel le LIKE (comparaison d'une chaîne à une expression régulière), et bien d'autres...

3.4.2 IN

```
SELECT num_emp, nom FROM Emp WHERE nom IN {'Dupond','Dupont'};
```

Sous cette forme ce n'est qu'une abréviation pour

```
SELECT num_emp, nom FROM Emp
WHERE nom = 'Dupond' OR nom = 'Dupont';
```

Mais IN peut aussi vérifier une appartenance à un **SELECT imbriqué** :

```
SELECT num_emp, nom, prenom FROM Emp WHERE dept IN
  (SELECT num_dept FROM Dept WHERE num_chef IN
     (SELECT num_emp FROM Emp WHERE nom = 'Lorenz'));
```

Cette dernière requête est équivalente à la jointure

```
SELECT E.num_emp, E.nom, E.prenom FROM Emp E, Dept, Emp C
WHERE E.dept = num_dept
AND num_chef = C.num_emp AND nom = 'Lorenz';
```

3.4.3 Autres prédicats du second ordre

En vrac :
- EXISTS (SELECT...) prend la valeur *vrai* si et seulement si le SELECT imbriqué donne au moins un n-uplet.
- (SELECT...) CONTAINS (SELECT...) est vrai si et seulement si le premier SELECT imbriqué contient tous les n-uplets du second.
- Comme le IN (cf. NOT IN), ces deux prédicats peuvent être niés pour donner les prédicats NOT EXISTS et NOT CONTAINS...

Remarque La valeur d'un SELECT imbriqué est indépendante de l'endroit où il est placé. Comme dans une jointure, les liens entre le SELECT imbriqué et son environnement doivent être explicites (ce qui peut imposer de renommer).

3.4.4 Fonctions du premier ordre

Les fonctions arithmétiques classiques +, -, *, / – ainsi que d'autres fonctions du premier ordre – peuvent être utilisées à peu près partout où interviennent des valeurs : clause SELECT, clause WHERE, clause ORDER BY... Ainsi :

```
SELECT num_emp, nom, salaire + prime FROM Emp
WHERE salaire <= prime * 3
ORDER BY salaire + prime;
```

3.4.5 Fonctions du second ordre

De même que les prédicats, certaines fonctions prennent pour argument un SELECT : ce sont ALL (tout n-uplet) et ANY (au moins un élément) :

- Employés touchant plus que n'importe quel informaticien :
  ```
  SELECT * FROM Emp WHERE salaire >= ALL
    (SELECT salaire FROM Emp WHERE dept = 'Info');
  ```
- Employés touchant plus qu'au moins un informaticien :
  ```
  SELECT * FROM Emp WHERE salaire >= ANY
    (SELECT salaire FROM Emp WHERE dept = 'Info');
  ```

3.5 Fonctions d'agrégation (group functions)

Un cas particulier de fonction du second ordre est représenté par les fonctions d'agrégation. Leur usage n'est pourtant pas le même.

Premier cas, celui d'un calcul sur toute une table :

```
SELECT COUNT(*), AVG(salaire) FROM Emp WHERE dept = 'Info';
```

donne le nombre d'employés du département d'informatique et la moyenne de leurs salaires.

Les fonctions d'agrégation sont COUNT (compter), SUM (somme), AVG (moyenne) , MIN, MAX, etc... Elles disposent toutes d'une option DISTINCT, qui n'a de sens réel qu'appliquée à la fonction COUNT.

```
SELECT COUNT(dept) FROM Emp;
```

compte les employés qui ont un département (quand *dept* prend la valeur NULL le n-uplet n'est pas compté).

```
SELECT COUNT(DISTINCT dept) FROM Emp;
```

compte les départements de noms différents.

3.5.1 Clause GROUP BY

Lorsque l'on veut obtenir une liste de tels calculs, il est nécessaire de **grouper** les n-uplets par les valeurs identiques des attributs du SELECT :

```
SELECT dept, AVG(salaire) FROM Emp GROUP BY dept;
```

Remarque De fait SQL exige que *tous* les attributs présents dans le SELECT (hormis naturellement ceux placés dans une fonction d'agrégation) soient aussi dans le GROUP BY. Ainsi, si l'on désire ajouter à la requête précédente le budget du département :

```
SELECT dept, sum(salaire), budget FROM Emp, Dept
WHERE dept = nom_dept GROUP BY dept, budget;
```

Logiquement, grouper par *budget* ne sert à rien si un département n'a qu'un budget. Mais informatiquement, quand on groupe par *dept*, les autres attributs, dont *budget*, ne sont plus accessibles.

3.5.2 Clause HAVING

Les conditions utilisant une fonction d'agrégation doivent être placées dans une clause HAVING (et **non dans une clause WHERE**) :

```
SELECT dept FROM Emp WHERE salaire > 15000
GROUP BY dept HAVING count(*) > 4;
```

Le WHERE étant évalué avant le GROUP BY, cette requête renvoie la liste des départements dont au moins cinq employés ont un salaire supérieur à 15000.

3.6 NULL

On l'a vu, SQL permet l'utilisation d'une valeur nulle NULL. Lorsque cette valeur est argument d'une fonction ou d'un prédicat, ces derniers prennent la valeur NULL. Ainsi 45 + NULL prend la valeur NULL, de même que les tests 67 = NULL, NULL = NULL ou NOT NULL (N.B. : Dans un test, NULL est faux[1]). Cette valeur nulle peut être manipulée à l'aide du prédicat IS ou de la fonction NVL (Null Valued Like) : NVL(attribut,valeur) prend la valeur *attribut* si cette dernière n'est pas NULL, et *valeur* sinon. Voir aussi le OUTER JOIN.

```
SELECT * FROM Emp WHERE dept IS NULL;
```

donne la liste des employés qui n'ont pas de département

```
SELECT num_emp, salaire + NVL(prime, salaire / 10) FROM Emp;
```

associe à chaque employé la somme de ses *salaire* et *prime*, en considérant une prime NULL comme égale à 10 pour cent du salaire.

Remarque Ne pas confondre NULL avec la valeur 0[2]. Une prime NULL ne signifie pas que l'employé n'en touche pas, mais que l'attribut *prime* ne s'applique pas. Par exemple, le dit employé n'étant pas un vendeur, il n'y a rien sur quoi faire porter une "commission sur les ventes", ce qui n'empêche pas que, pour des raisons syndicales ou de simple justice, il touche un complément de salaire, fonction de son salaire, mais que l'on ne peut appeler *prime*.

3.7 Opérateurs "ensemblistes"

Les opérateurs ensemblistes de l'algèbre sont simulés par UNION (**réunion**), MINUS (**différence**) et INTERSECTS (**intersection**). Les SELECT sur lesquels ils portent doivent être union-compatibles.

```
SELECT nom_dept FROM Dept MINUS SELECT dept FROM Emp;
```

[1] ce qui implique les seules exceptions à cette règle **d'absorption du NULL** : "NULL ou vrai" est vrai ; quand "NULL et faux" est faux, et pas NULL : "non(NULL et faux)" est vrai.

[2] Dans les chaînes de caractères, NULL est cependant identifié à la chaîne vide.

donne les départements qui ne figurent pas dans Emp, soit ceux qui n'ont pas d'employés. C'est l'une des deux manières standard de réaliser une négation. L'autre, plus proche du calcul relationnel, utilise le NOT IN :

```
SELECT nom_dept, budget FROM Dept
WHERE dept NOT IN (SELECT dept FROM Emp);
```

Cette dernière méthode permet d'ajouter d'autres informations (ici le budget), ce qui serait plus compliqué dans la première version.

4 Autres fonctionnalités

Je donne ici, en vrac et sans souci de présenter le formalisme exact – qui varie énormément –, un résumé des autres fonctionnalités de SQL – aussi bien, du reste, que de tout SGBD convenablement implémenté.

4.1 Confidentialité

Chaque utilisateur dispose d'un nom de compte, et d'un mot de passe.

- Du point de vue du SGBD, les droits sont CONNECT (droit d'utiliser SQL), RESOURCE (droit de créer des tables) et DBA (pour DataBase Administrator : l'administrateur de la base de données).
- Par ailleurs l'utilisateur peut avoir des droits spécifiques sur une table. Ces droits sont SELECT (droit de la consulter), INSERT (droit d'y ajouter des n-uplets), UPDATE (droit d'y modifier des n-uplets – ce droit peut être précisé, pour qu'il ne puisse y modifier que certains attributs), DELETE (droit d'y supprimer des n-uplets), ALTER (droit d'en modifier la définition) et GRANT (droit de donner des droits à d'autres utilisateurs, à l'aide de la commande GRANT, que je ne détaille pas). Le créateur d'une table (tout comme le DBA) a tous les droits sur celle-ci.

4.2 Optimisation

SQL admet une optimisation implicite. Il est cependant possible de rendre plus rapides certaines jointures par la définition d'index, ce qui, par ailleurs, ralentit les mises-à-jour. Par exemple,

```
CREATE INDEX I_emp ON Emp.num_emp;
```

(ou une écriture analogue) créera un index sur l'attribut *num_emp* de la table Emp. Cela rendra plus efficace l'accès à la table *Emp* via l'attribut *num_emp*, et donc les jointures sur cet attribut. Par contre l'ajoût d'un employé nécessitera parfois une réorganisation de l'index, et prendra donc plus de temps.

SQL permet aussi de créer un UNIQUE INDEX, qui impose que l'attribut indexé ne présente pas de duplicata. Ce sera donc une **clef** de la relation.

4.3 Présentation : menus, rapports et formulaires

Ces fonctionnalités sont destinées à l'utilisateur final. Les manipulations utiles (**traitements**) sont interfacées pour qu'il n'ait pas à passer par SQL.

- Un **rapport** est la présentation "élégante" d'un résultat de requête (un bulletin de salaire est une forme de rapport).
- Un **formulaire** permet de réaliser des mises-à-jour en remplissant les cases d'un écran.
- Enfin un **menu** permet de naviguer entre ces différents traitements.

Ces outils sont généralement conçus à l'aide de logiciels étendant SQL proprement dit (chez ORACLE : SQL-report, SQL-forms et SQL-menu) ; cependant SQL permet de définir des commandes paramétrées (donc analogues à un formulaire) ou de présenter quelque peu le résultat (taille des colonnes, etc. . . : présentation liée aux rapports). La notion de **vue** a également un lien avec ces fonctionnalités.

4.4 Intégration à un langage

Les possibilités de SQL peuvent être augmentées en l'intégrant à un autre langage, soit défini par le constructeur (INGRES, ORACLE,. . . proposent des langages de quatrième génération adaptés à SQL), soit plus universel (C, Pascal, COBOL, etc. . .). La difficulté est alors d'intégrer un langage déclaratif (SQL) à un autre, généralement très peu déclaratif. . . Mais encore une fois, ceci n'a pas sa place dans un "petit" manuel SQL.

Index