Date de dernière modification *: 15/06/2015*

Annexe **24**

SQL, les ensembles et la logique

Le domaine des bases de données utilise de manière intensive des éléments de la théorie des ensembles et de la logique. SQL en particulier a été élaboré à partir de ces deux branches des mathématiques. Ce chapitre rassemble quelques éléments qui apparaissent explicitement dans l'écriture des requêtes. On étudiera successivement quelques principes utiles relatifs aux ensembles, aux multi-ensembles et à la logique. Pour cette dernière, on décrira les opérateurs de la logique ternaire.

A24.1SQL ET LES MATHÉMATIQUES

Le sous-langage SQL-DML nous permet de formuler des requêtes dont le résultat se présente sous la forme d'une table, laquelle contient une *collection* de lignes. Sous certaines conditions, cette collection est un *ensemble* de lignes, c'est-à-dire qu'elle ne contient pas deux lignes identiques. Les requêtes produisent le plus souvent des ensembles de données à partir d'autres ensembles via des formules *logiques* qu'on exprime dans les clauses where.

La présence de concepts mathématiques dans le langage SQL n'est évidemment pas un hasard : ce langage a été construit sur des branches des mathématiques telles que l'algèbre relationnelle et le calcul relationnel, ce dernier étant une variante de la logique classique.

Certaines traces de cette origine sont assez visibles. Quelques exemples :

1. Il est possible de construire une table à partir de deux autres tables à l'aide des opérateurs ensemblistes d'*union*, d'*intersection* et de *différence*.

2. La clause where d'une requête select-from-where comporte une condition qui peut être de complexité quelconque, formée de conditions plus élémentaires assemblées à l'aide des opérateurs and, or et not.
3. La fonction exists permet de vérifier l'existence d'éléments qui satisfont une condition déterminée. Elle rappelle le quantificateur existentiel de la logique.
4. La clause from permet de définir, via les alias de table, des variables qui prennent leur valeur parmi les lignes d'une table, et qui sont utilisées dans les conditions de la clause where. Nous voici très proches des variables de prédicats.

Il n'est sans doute pas inutile de rafraîchir quelque peu la mémoire du lecteur en rappelant quelques éléments de la théorie des ensembles et de la logique. Pas de couverture complète du sujet donc, mais une révision rapide des éléments directement utilisés dans la pratique des bases de données.

A24.2ENSEMBLE, C'EST TOUT

A24.2.1Ensemble

Un ensemble est une collection d'objets distincts, appelés *éléments* de l'ensemble. Un ensemble est défini

- en *extension* (on cite tous ses éléments dans un ordre quelconque)
- ou en *compréhension* (on précise les propriétés communes à tous les éléments).

En toute généralité, un ensemble est défini par une formule du type $\{x: P(x)\}$, où P est un prédicat aussi appelé *fonction caractéristique* de l'ensemble. Un ensemble est appelé *singleton* s'il contient un seul élément et *paire* s'il en contient deux. L'*ensemble vide* ne contient aucun élément. Il se note { } ou ∅.

Exemples

- $A = \{2, 4, 6, 8\}$; l'ensemble A, défini en *extension*, est formé des entiers naturels pairs inférieurs à 10.
- $B = \{x: (x \in \mathbb{N}) \wedge (10 \leq x \leq 100)\}$; l'ensemble B, défini en *compréhension*, est formé des entiers naturels de 10 à 100. Cette définition peut aussi s'écrire de manière plus synthétique : $B = \{x \in \mathbb{N}: 10 \leq x \leq 100\}$.
- Le contenu d'une table d'une base de données relationnelle munie d'un identifiant est un *ensemble* de lignes. Le domaine d'une colonne est constitué d'un *ensemble* de valeurs, défini en extension ou en compréhension.

A24.2.2Appartenance à un ensemble

L'objet a appartient à l'ensemble A, ce qu'on note $a \in A$, si a est un élément de A. Si P est la fonction caractéristique de A, alors il est équivalent de dire que P(a) est vrai, ou plus simplement P(a).

Exemple

- le prédicat "in" en SQL correspond à une relation d'appartenance. Soit la condition NCLI **in** (select NCLI from COMMANDE) évaluée pour une ligne r; elle est évaluée à vrai si r.NCLI ∈ E, où E dénote l'ensemble des valeurs de NCLI présentes dans la table COMMANDE. On notera que l'évaluation de l'expression (select NCLI from COMMANDE) produit en réalité un multi-ensemble.

A24.2.3Inclusion d'un ensemble dans un autre

L'ensemble A est inclus dans l'ensemble E, ce qu'on note $A \subseteq E$, si tous les éléments de A sont aussi des éléments de E. L'inclusion est dite *stricte* si certains éléments de E n'appartiennent pas à A. On la note alors $A \subset E$. Si P_A est la fonction caractéristique de A, et P_E est la fonction caractéristique de E, alors $A \subseteq E \equiv \forall a, P_A(a) \Rightarrow P_E(a)$. Si A est inclus dans E, on dira aussi que A est un sous-ensemble, ou une partie, de E. *Note* : l'ensemble vide est inclus dans tout autre ensemble.

En SQL

Cherchons à déterminer si l'ensemble des valeurs de la colonne NCLI de COMMANDE est inclus dans celui des valeurs de la colonne NCLI de la table CLIENT (comme si, par négligence, on avait omis de déclarer cette colonne foreign key). Pour ce faire, recherchons les lignes de COMMANDE pour lesquelles cette relation d'inclusion n'est pas respectée :

```
select *
from   COMMANDE M
where  not exists(select NCLI from CLIENT
                  where  NCLI = M.NCLI);
```

A24.2.4Taille ou cardinal d'un ensemble

Le nombre d'éléments d'un ensemble est appelé *taille* ou *cardinal* de cet ensemble. Le cardinal de A se note |A| ou card(A). Quelques relations utiles :

- $|A1 \cup A2| = |A1| + |A2| - |A1 \cap A2|$
- $|A1 - A2| = |A1| - |A1 \cap A2|$

Exemples

- $|\{2, 7, 6, 8\}| = 4$
- $|\varnothing| = 0$

En SQL

C'est la fonction agrégative count qui va nous renseigner sur la taille de divers ensembles (ou multi-ensembles) :

```
select count(*)
from   DETAIL
where  NPRO = 'PA60';
```

A24.2.5Égalité de deux ensembles

Deux ensembles A_1 et A_2 sont égaux, ce qu'on note $A_1 = A_2$, si tout élément de l'un appartient à l'autre et inversement. On a aussi :

$(A_1 = A_2) \Leftrightarrow (A_1 \subseteq A_2 \wedge A_2 \subseteq A_1)$.

L'égalité de deux ensembles peut également être vérifiée par la propriété suivante :

$(A_1 = A_2) \Leftrightarrow ((A_1 \cup A_2) = (A_2 \cap A_1))$

Cette propriété est plus aisée à évaluer en SQL : les ensembles A_1 et A_2 sont égaux si l'expression suivante renvoie une table vide :

$(A_1 \cup A_2) - (A_2 \cap A_1)$

En SQL

Considérons deux tables T1(C) et T2(C). Ces deux tables sont égales si la requête suivante renvoie un résultat vide :

```
(select C from T1 union select C from T2)
 minus
(select C from T1 intersect select C from T2);
```

La section B.4.6 suggère une autre manière de vérifier l'égalité de deux ensembles A et B : on vérifie que A est inclus dans B et que A et B ont la même taille.

A24.2.6Union de deux ensembles

L'union de deux ensembles A_1 et A_2 est un ensemble E qui contient tous les éléments de A_1 et de A_2, et eux seulement. On la note $E = A1 \cup A2$. Le terme *union* désigne aussi l'opérateur $\cup$ qui produit E. Si P_1 est la fonction caractéristique de A_1, et P_2 est la fonction caractéristique de A_2, E est l'union de A_1 et A_2 *ssi* $E = \{a: P_1(a) \vee P_2(a)\}$.

Exemples

- L'union $A \cup B$ des ensembles A = {a,b,c,d} et B = {c,d,e} produit l'ensemble AB = {a,b,c,d,e}.
- l'opérateur **union** de SQL construit l'union ensembliste des contenus de deux tables; si ces contenus sont des multi-ensembles (voir B.3.1), le résultat est néanmoins un ensemble.

En SQL

L'union des deux tables T1(C) et T2(C) est obtenue comme suit :

```
select C from T1
union
select C from T2;
```

A24.2.7Intersection de deux ensembles

L'intersection de deux ensembles A_1 et A_2 est un ensemble E qui contient tous les éléments qui appartiennent simultanément à A_1 et A_2, et eux seulement. On la note : $E = A_1 \cap A_2$. Le terme *intersection* désigne aussi l'opérateur $\cap$ qui produit E.

On notera que l'intersection n'est pas un opérateur primitif. En effet :

$A_1 \cap A_2 = A_1 - (A_1 - A_2) = A_2 - (A_2 - A_1)$

Si P_1 est la fonction caractéristique de A_1, et P_2 est la fonction caractéristique de A_2, E est l'intersection de A_1 et A_2 *ssi* $E = \{a: P_1(a) \wedge P_2(a)\}$.

Exemples

- L'intersection $A \cap B$ des ensembles $A = \{a,b,c,d\}$ et $B = \{c,d,e\}$ produit l'ensemble $AB = \{c,d\}$
- l'opérateur **intersect** de SQL construit l'intersection ensembliste des contenus de deux tables; si ces contenus sont des multi-ensembles, le résultat est néanmoins un ensemble.

En SQL

L'intersection des deux tables T1(C) et T2(C) est obtenue comme suit :

```
select C from T1
intersect
select C from T2;
```

on peut aussi obtenir l'intersection par une jointure :

```
select C
from   T1, T2
where  T1.C = T2.C;
```

A24.2.8Différence entre deux ensembles

La différence de deux ensembles A_1 et A_2 est un ensemble E qui contient tous les éléments de A_1 qui n'appartiennent pas à A_2, et eux seulement. On la note : $E = A_1 - A_2$ ou $A_1 \setminus A_2$.Le terme *différence* désigne aussi l'opérateur – qui produit E.
Si P_1 est la fonction caractéristique de A_1, et P_2 est la fonction caractéristique de A_2, E est la différence de A_1 et A_2 *ssi* $E = \{a: P_1(a) \wedge \neg P_2(a)\}$.

Exemples

- La différence $A - B$ des ensembles $A = \{a,b,c,d\}$ et $B = \{c,d,e\}$ produit l'ensemble $AB = \{a,b\}$.
- L'opérateur minus ou except de certains dialectes de SQL construit la différence ensembliste des contenus de deux tables; si ces contenus sont des multi-ensembles, le résultat est néanmoins un ensemble.

En SQL

La différence des deux tables T1(C) et T2(C) est obtenue comme suit :

```
select C from T1
minus
select C from T2;
```

on peut aussi obtenir la différence par une sous-requête :

```
select C
from   T1
where  C not in (select C from T2);
```

A24.2.9Puissance d'un ensemble

Soient E un ensemble. L'ensemble P de tous les sous-ensembles (ou parties) de E s'appelle l'*ensemble puissance* de E ou encore l'*ensemble des parties* de E. Il se note $P = 2^E$ ou P_E ou $P(E)$.

Exemples

- Soit E = {1,2,3}. L'ensemble puissance de E est le suivant (on notera la présence de l'ensemble vide) :

  $P(E)$ = {{},{1},{2},{3},{1,2},{2,3},{1,3},{1,2,3}}

A24.2.10Partition d'un ensemble

Soient E un ensemble non vide et P un ensemble de sous-ensembles de E (une partie de $P(E)$). P constitue une partition de E si :

1. l'union des éléments de P est égale à E,
2. les éléments de P sont disjoints deux à deux,
3. P ne contient pas l'ensemble vide.

Exemples

- Soit E = {1,2,3,4,5}. Les trois ensembles suivants sont des partitions de E :

  P1 = {{1,2},{3},{4,5}}
  P2 = {{1,2}{3,4,5}}
  P3 = {{1},{2},{3},{4},{5}}

A24.2.11Propriétés des opérateurs ensemblistes

Les propriétés suivantes peuvent nous être très utiles dans la formulation de requêtes SQL :

- L'union et l'intersection sont commutatives[1] :

  $A_1 \cup A_2 = A_2 \cup A_1$ *et* $A_1 \cap A_2 = A_2 \cap A_1$

- L'union et l'intersection sont associatives[2] :

  $A_1 \cup (A_2 \cup A_3) = (A_1 \cup A_2) \cup A_3$ *et* $A_1 \cap (A_2 \cap A_3) = (A_1 \cap A_2) \cap A_3$

- L'union est distributive par rapport à l'intersection[3] :

  $A_1 \cup (A_2 \cap A_3) = (A_1 \cup A_2) \cap (A_1 \cup A_3)$

- L'intersection est distributive par rapport à l'union :

  $A_1 \cap (A_2 \cup A_3) = (A_1 \cap A_2) \cup (A_1 \cap A_3)$

- L'intersection est distributive par rapport à la différence :

1. Un opérateur binaire ω est *commutatif* si, quels que soient les arguments x et y, on a : $x \omega y = y \omega x$
2. Un opérateur binaire ω est *associatif* si, quels que soient les arguments x, y et z on a : $x \omega (y \omega z) = (x \omega y) \omega z$
3. Un opérateur binaire ω est *distributif* par rapport à un opérateur binaire ϕ si, quels que soient les arguments x, y et z on a : $x \omega (y \phi z) = (x \omega y) \phi (x \omega z)$

$$A_1 \cap (A_2 - A_3) = (A_1 \cap A_2) - (A_1 \cap A_3)$$

En revanche :

- La différence **n'est pas** commutative :

  $$A_1 - A_2 \neq A_2 - A_1$$

- La différence **n'est pas** associative :

  $$A_1 - (A_2 - A_3) \neq (A_1 - A_2) - A_3$$

- L'union **n'est pas** distributive par rapport à la différence :

  $$A_1 \cup (A_2 - A_3) \neq (A_1 \cup A_2) - (A_1 \cup A_3)$$

- La différence **n'est pas** distributive par rapport à l'union :

  $$A_1 - (A_2 \cup A_3) \neq (A_1 - A_2) \cup (A_1 - A_3)$$

- La différence **n'est pas** distributive par rapport à l'intersection[4] :

  $$A_1 - (A_2 \cap A_3) \neq (A_1 - A_2) \cap (A_1 - A_3)$$

A24.3LES MULTI-ENSEMBLES

Toute table qui ne fait pas l'objet d'une contrainte d'unicité n'est pas un ensemble mais un multi-ensemble. Il en est parfois ainsi en particulier de résultats de requêtes ou du contenu de vues. Il est donc particulièrement utile d'étudier les propriétés et les moeurs reproductives de ces collections qui s'avèrent largement plus laxistes que les ensembles[5].

A24.3.1Notion de multi-ensemble

Un multi-ensemble est une collection d'éléments non nécessairement distincts. Pour les étudier, nous admettrons qu'un multi-ensemble est équivalent à un ensemble de couples dont le premier composant indique un élément du multi-ensemble, et le second la multiplicité du premier, indiquant le nombre d'occurrences de cet élément qui apparaissent dans le multi-ensemble. Un ensemble est un multi-ensemble dont toutes les multiplicités sont à 1.

Exemples

- Le multi-ensemble M = {2, 4, 2, 7} est formé de 4 éléments; il peut être défini par l'ensemble suivant : {(4,1), (2,2), (7,1)}.
- Les diviseurs premiers de 20 forment le multi-ensemble {1, 2, 2, 5}.
- Le contenu d'une table d'une base de données relationnelle à laquelle n'est associé aucun identifiant est un multi-ensemble de lignes.

4. Sauf dans la configuration particulière de la seconde loi de de Morgan, où A_2 et A_3 sont des parties de A_1.

5. Voir en particulier : Syropoulos, Apostolos, Mathematics of Multisets, in C. S. Calude et al., eds., *Multiset processing: Mathematical, computer science, and molecular computing points of view*, LNCS 2235, 2001, Springer-Verlag

- Le résultat d'une requête SQL sans clause group by, et dont la liste select ne comprend pas tous les composants d'un identifiant ni la clause distinct est un multi-ensemble de lignes. *Exemple* : select CAT, LOCALITE from CLIENT ne renvoie pas un ensemble mais un multi-ensemble de lignes.

A24.3.2Appartenance à un multi-ensemble

Etant donné un objet m et un multi-ensemble M, m appartient à M, ce qu'on note $m \in M$, si m est présent au moins une fois dans M.

A24.3.3Inclusion d'un multi-ensemble dans un autre

Le multi-ensemble M est inclus dans le multi-ensemble Q, ce qu'on note $M \subseteq Q$, si chaque élément de M présent en n_m exemplaires est présent dans Q en n_q exemplaires, avec $n_m \leq n_q$.

A24.3.4Union de deux multi-ensembles

L'union de deux multi-ensembles M_1 et M_2 est un multi-ensemble M qui contient tous les éléments de M_1 et tous ceux de M_2. La multiplicité dans M d'un élément est $n_1 + n_2$, où n_1 est la multiplicité de l'élément dans M_1 et n_2 ce nombre dans M_2. On note l'union $M = M_1 \cup M_2$. Le terme *union* désigne aussi l'opérateur qui produit M.

Exemples

- L'union $A \cup B$ des multi-ensembles A = {1,2,2,4} et B = {2,4,5} produit le multi-ensemble
  AB = {1,2,2,2,4,4,5}.
- L'opérateur SQL union all représente l'union des contenus de deux tables sans réduction des lignes en double. Même si le contenu de chaque table est un ensemble de lignes, le résultat peut être un multi-ensemble. En effet toute ligne présente dans les deux tables apparaît deux fois dans l'union.

En SQL

L'union des deux tables T1(C) et T2(C), considérées comme des multi-ensembles, est obtenue comme suit :

```
select C from T1
union all
select C from T2;
```

A24.3.5Intersection de deux multi-ensembles

Opérateur qui, à partir de deux multi-ensembles M_1 et M_2, produit un multi-ensemble M qui contient tous les éléments qui appartiennent simultanément à M_1 et M_2. La multiplicité d'un élément est $\min(n_1,n_2)$, où n_1 est la multiplicité de l'élément dans M_1 et n_2 ce nombre dans M_2. On note l'intersection $M = M_1 \cap M_2$. Le terme *intersection* désigne aussi l'opérateur qui produit M.

Exemples

- L'intersection $A \cap B$ des multi-ensembles A = {1,2,2,4} et B = {2,4,5} produit le multi-ensemble
  AB = {2,4}.
- L'opérateur SQL intersect all représente l'intersection des contenus de deux tables sans réduction des lignes en double. Ses arguments et son résultat peuvent donc être des multi-ensembles.

En SQL

L'intersection des deux tables T1(C) et T2(C), considérées comme des multi-ensembles, est obtenue comme suit :

```
select C from T1
intersect all
select C from T2;
```

A24.3.6Différence entre deux multi-ensembles

Opérateur qui, à partir de deux multi-ensembles M_1 et M_2, produit un multi-ensembles M qui contient tous les éléments de M_1 qui y apparaissent en plus grand nombre que dans M_2. La multiplicité d'un élément retenu est $n_1 - n_2$, où n_1 est la multiplicité de l'élément dans M_1 et n_2 ce nombre dans M_2. On le note $M = M_1 - M_2$. Le terme *différence* désigne aussi le résultat de l'application de cet opérateur, soit le multi-ensemble M.

Exemples

- La différence $A - B$ des multi-ensembles A = {1,2,2,4} et B = {2,4,5} produit le multi-ensemble
  AB = {1,2}.
- L'opérateur SQL except all représente la différence des contenus de deux tables sans réduction des lignes en double. Ses arguments et son résultat peuvent donc être des multi-ensembles.

En SQL

La différence des deux tables T1(C) et T2(C), considérées comme des multi-ensembles, est obtenue comme suit :

```
select C from T1
except all
select C from T2;
```

A24.3.7Synthèses des opérateurs SQL

Le tableau ci-dessous résume le fonctionnement des six opérateurs SQL pour un même élément présent respectivement en n1 et n2 exemplaires dans chaque multi-ensemble.

	multi-ensemble 1	multi-ensemble 2	résultat
union	n1	n2	max(min(n1,1), min(n2,1))
union all	n1	n2	n1+n2
intersect	n1	n2	min(n1,1) x min(n2,1)
intersect all	n1	n2	min(n1,n2)
except	n1	n2	min(max(n1-n2,0),1)
except all	n1	n2	max(n1-n2,0)

A24.3.8Opérateurs algébriques sur les multi-ensembles

La projection, la jointure et la sélection peuvent être étendues aux multi-ensembles, de manière à correspondre au comportement de SQL lorsqu'une requête est appliquée à une table, réelle ou virtuelle, susceptible de comporter des lignes en double. On trouvera dans [Garcia-Molina, 2008] une présentation détaillée de ces opérateurs.

A24.4UN PEU DE LOGIQUE

Dans le langage SQL, les expressions logiques apparaissent principalement dans la clause where des requêtes select-from-where, update et delete, mais on les trouvera aussi dans la déclaration des vues, dans les contraintes de type check, dans la clause conditionnelle des triggers, dans le corps des *procédures SQL* ou encore, en SQL-1999, dans des sous-expressions des clauses from et select. Au delà du langage SQL, les expressions logiques interviendront encore dans la théorie relationnelle, dans l'expression de contraintes dans les schémas conceptuels et dans langages de contraintes tels que OCL d'UML.

Il est donc particulièrement important de rappeler quelques principes et propriétés des expressions conditionnelles, qui traduisent le plus souvent des formules de la logique classique mais aussi non classique telle que la logique ternaire.

Un autre argument milite en faveur d'une section consacrée à la logique, c'est la distance, voire l'antagonisme parfois, qui existe entre la logique que nous pratiquons dans notre vie de tous les jours et la logique formelle qui est celle d'SQL et des langages de programmation. Les opérateurs élémentaires et, ou et non n'y ont pas toujours le même sens[6] !

A24.4.1Propositions, prédicats et conditions

Rappelons d'abord que la logique classique traite des énoncés auxquels on assigne, directement ou non, la valeur vrai ou la valeur faux. Ces deux valeurs sont appelées *valeurs de vérité*. SQL, plus gourmand, en exigera une troisième : inconnu.

6. Lire par exemple [Cuénod, 1999]

Proposition

Une proposition est un énoncé (une affirmation, une phrase quelconque) auquel est assignée une valeur de vérité. Cette assignation est arbitraire, la logique ne se préoccupant pas de la validité de cet énoncé. Si nous assignons la valeur vrai à l'énoncé "tous les hommes sont rouges", nous obtenons une proposition parfaitement valide. Ceci ne doit pas nous choquer : introduire dans la table DETAIL la ligne (30264, PA45, 8) ne va pas perturber significativement la validité de la base de données, même si nous apprenons plus tard que ces données sont fausses et doivent être remplacées par (30264, PA45, 18). Cette ligne n'est rien d'autre qu'une proposition[7] à laquelle nous avons assigné la valeur vrai en la rangeant dans la table. Si cette ligne ne se trouve pas dans la table, la proposition qu'elle représente a automatiquement la valeur faux. De telles erreurs d'introduction de données ne concernent en rien ni la logique formelle ni le SGBD SQL !

Prédicat

Un prédicat est un énoncé comportant une ou plusieurs variables et qui n'est ni vrai ni faux. Si on assigne une valeur à chacune de ses variables, alors le prédicat se transforme en une proposition à laquelle on peut attribuer une valeur de vérité.

"X est un nombre premier" est un prédicat. Il n'est ni vrai ni faux tant que je ne connais pas la valeur de X. Si j'y remplace X par un nombre entier, alors je peux lui assigner une valeur de vérité :

- "3 est un nombre premier" = vrai
- "4 est un nombre premier" = faux

Prédicats, propositions et SQL

Dans une requête SQL, si CLI est un alias de table, alors l'expression "CLI.CAT is not null", qu'on peut trouver dans une clause where, est un prédicat. En effet, CLI.CAT y est une variable puisque CLI peut prendre comme valeur n'importe quelle ligne de la table CLIENT. C'est lorsque cette expression sera évaluée pour une ligne particulière de CLIENT, ligne désignée par CLI, ou ligne courante, que l'expression devient une proposition. Si elle vaut vrai, alors la ligne sera retenue sinon, elle sera ignorée. Dans la plupart des cas[8], une condition SQL est un prédicat.

Si une ligne d'une table représente une proposition, alors on peut admettre que le schéma de la table représente un prédicat. A la table DETAIL je peux associer le prédicat "la commande n° NCLI a commandé le produit n° NPRO en quantité QCOM" ou, de manière plus compacte DETAIL(NCOM, NPRO, QCOM). En remplaçant chacune de ces trois variables par une valeur, j'obtiens une ligne qui représente une proposition. Si cette ligne se trouve dans la table, c'est qu'on a décidé que cette proposition a la valeur vrai, sinon elle a la valeur faux. Cette interprétation s'appelle

7. En effet, elle traduit de manière compacte l'énoncé "la commande n° 30264 a commandé le produit n° PA45 en quantité 8".
8. L'expression where 1 = 2, parfaitement valable en SQL, comporte la proposition "1 = 2" dont on peut affirmer sans hésitation qu'elle aura toujours la valeur faux, quelle que soit la ligne pour laquelle elle est évaluée !

l'*hypothèse du monde fermé* : *ce qui est représenté dans la base de données est vrai, ce qui ne l'est pas est faux*[9].

Nous examinerons à la section B.4.6 une autre manière de transformer un prédicat en proposition.

A24.4.2Expressions composées : opérateurs logiques et, ou, non

Il est souvent nécessaire de construire une condition (proposition ou prédicat) à partir de conditions plus élémentaires. Outre la *négation* (communément notée ¬ et dénommée `not` en SQL), deux opérateurs logiques à deux arguments nous permettront de construire des conditions composées. Désignons par P et Q deux conditions quelconques, élémentaires ou composées.

- la *conjonction*, notée P ∧ Q (P et Q en français, P and Q en SQL); cette expression vaut vrai si les deux arguments sont vrais, et faux dans tous les autres cas
- la *disjonction*, notée P ∨ Q (P ou Q en français, P or Q en SQL); cette expression vaut vrai si au moins un des arguments est vrai, et faux dans l'autre cas.

Lors de l'évaluation d'une expression complexe, la négation (non) a priorité sur la conjonction (et), qui elle-même a priorité sur la disjonction (ou). On peut imposer un ordre d'évaluation différent à l'aide de parenthèses. Soit, par exemple, l'expression SQL :

```
LOCALITE = 'Toulouse' and COMPTE < 0 or CAT = 'C1'
```

Elle indique qu'une ligne de CLIENT est sélectionnée si (LOCALITE = 'Toulouse' et COMPTE < 0) ou encore si (CAT = 'C1') ou si les deux conditions sont vérifiées. On aurait pu écrire, de manière équivalente mais plus claire :

```
(LOCALITE = 'Toulouse' and COMPTE < 0) or CAT = 'C1'
```

En revanche, l'expression :

```
LOCALITE = 'Toulouse' and (COMPTE < 0 or CAT = 'C1')
```

a une tout autre interprétation : une ligne de CLIENT est sélectionnée si elle vérifie simultanément deux conditions : d'une part, (LOCALITE = 'Toulouse'), et d'autre part, (COMPTE < 0, ou bien CAT = 'C1', ou encore les deux conditions simultanément). Il sera prudent, même lorsqu'elles ne sont pas strictement nécessaires, d'user de paren-

9. L'*hypothèse du monde ouvert* stipule que *ce qui n'est pas dans la base de données n'est ni faux, ni vrai, mais est inconnu*. Ce modèle est conceptuellement plus réaliste et plus puissant, mais techniquement irréalisable dans la plupart des cas. Il faudrait en effet enregistrer ce que nous savons être vrai mais aussi ce que nous savons être faux. Est dès lors inconnu ce qui n'a pas été explicitement ou indirectement (par inférence) affirmé vrai ou faux. Si nous savons que *GILLET a commandé les produits CS464, PA45, PA60 et PH222*, il faudrait aussi enregistrer *les faits qu'il n'a pas commandé les produits PS222, CS464 ni CS262 ...* , puisque ces faits négatifs sont connus.

thèses dans l'écriture d'expressions complexes afin d'éviter toute ambiguïté d'interprétation.

A24.4.3Opérateurs logiques additionnels

A l'occasion, trois autres opérateurs pourront s'avérer utiles :

- la *disjonction exclusive*, notée P ⊕ Q (ou exclusif en français ou XOR, absent d'SQL); cette expression est vraie si un et un seul des arguments est vrai;
- l'*implication*, notée P ⇒ Q (implique en français, absent d'SQL); cette expression est vraie sauf si P est vrai et Q est faux;
- l'*équivalence*, notée P ⇔ Q (équivaut à en français, absent d'SQL); cette expression est vraie lorsque P et Q ont la même valeur.

A24.4.4Tables de vérité binaires

Un opérateur logique ω est complètement défini par sa *table de vérité*, qui indique la valeur (vrai ou faux) de l'expression P ω Q pour chaque combinaison de valeurs des arguments P et Q. On rappelle ci-dessous les tables des opérateurs rencontrés jusqu'ici :

P	***Q***	¬ **P**	¬ **Q**	**P** ∧ **Q**	**P** ∨ **Q**	**P** ⊕ **Q**	**P** ⇒ **Q**	**P** ⇔ **Q**
vrai	***vrai***	faux	faux	vrai	vrai	faux	vrai	vrai
vrai	***faux***	faux	vrai	faux	vrai	vrai	faux	faux
faux	***vrai***	vrai	faux	faux	vrai	vrai	vrai	faux
faux	***faux***	vrai	vrai	faux	faux	faux	vrai	vrai

Son interprétation est la suivante : pour toute combinaison de valeurs de P et Q, la ligne correspondante donne la valeur de vérité de chaque opérateur. Par exemple, la deuxième ligne nous dit que

> si P = vrai et Q = faux,
> alors, en particulier, P ∨ Q = vrai et P ⇒ Q = faux.

A24.4.5Les équivalences remarquables

Soient P, Q et R trois conditions quelconques, élémentaires ou composées. La logique classique admet les équivalences suivantes, qu'on démontre aisément en vérifiant que les tables de vérité des deux membres sont identiques.

Un Ricard, sinon rien (principe du tiers exclu)

Il n'y pas d'autres valeurs que vrai et faux : ce qui n'est pas vrai est faux et inversement.

- ¬ `vrai` ≡ `faux` *r1*
- ¬ `faux` ≡ `vrai` *r2*

Le beurre et l'argent du beurre (complémentarité)[10]

- $(\neg P) \wedge P \equiv \text{faux}$ *r3*
- $(\neg P) \vee P \equiv \text{vrai}$ *r4*

Vous n'êtes pas sans savoir (double négation)

- $\neg (\neg P) \equiv P$ *r5*

Schtroumpf vert et vert Schtroumpf (commutativité)

- $P \vee Q \equiv Q \vee P$ *r6*
- $P \wedge Q \equiv Q \wedge P$ *r7*

Jules et Jim (associativité)[11]

- $(P \vee Q) \vee R \equiv P \vee (Q \vee R)$ *r8*
- $(P \wedge Q) \wedge R \equiv P \wedge (Q \wedge R)$ *r9*

Les Restos du coeur (distributivité)

- $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$ *r10*
- $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$ *r11*

Purification ethnique (lois de de Morgan)

- $\neg (P \wedge Q) \equiv (\neg P) \vee (\neg Q)$ *r12*
- $\neg (P \vee Q) \equiv (\neg P) \wedge (\neg Q)$ *r13*

Ariel ou Javel (éléments neutres et absorbants)[12]

- $P \wedge \text{vrai} \equiv P$ *r14*
- $P \wedge \text{faux} \equiv \text{faux}$ *r15*
- $P \vee \text{vrai} \equiv \text{vrai}$ *r16*
- $P \vee \text{faux} \equiv P$ *r17*

Les Faussaires

L'implication ($\Rightarrow$), la disjonction exclusive ($\oplus$) et l'équivalence ($\Leftrightarrow$) n'existent pas en SQL. Elles doivent donc être remplacées par des expressions équivalentes.

- $P \Rightarrow Q \equiv (\neg P) \vee Q$ *r18*
- $P \oplus Q \equiv (P \wedge \neg Q) \vee (\neg P \wedge Q)$ *r19*
- $P \oplus Q \equiv (P \vee Q) \wedge \neg (P \wedge Q)$ *r20*
- $P \Leftrightarrow Q \equiv (P \wedge Q) \vee \neg (P \vee Q)$ *r21*

10. Pour le sourire de la crémière, il faudra recourir à la logique ternaire, qui sera examinée à la section B.4.8.

11. Un peu de culture littéraire et cinématographique !

12. Celui-ci est sans doute un peu plus difficile. Un indice : *lequel est présumé préserver les couleurs alors que l'autre les détruit* ?

Ces tables et ces règles définissent la logique *binaire* (à deux valeurs, vrai et faux) ou *cartésienne* ou encore *du tiers exclu*. Nous verrons plus loin que certains aspects du langage SQL s'appuient également sur une logique ternaire, utilisant trois valeurs : vrai, faux et inconnu (section B.4.8). La plupart des règles ci-dessus ne sont plus d'application dans cette logique.

A24.4.6Quantificateurs existentiel et universel

Une proposition peut être obtenue en remplaçant chacune des variables d'un prédicat par une valeur déterminée. Il existe une autre manière de dériver une proposition à partir d'un prédicat : en réduisant une variable par un quantificateur existentiel (∃) ou universel (∀). Si toutes les variables sont réduites, par assignation de valeur ou par quantification, alors le prédicat se transforme en une proposition.

Un quantificateur permet de préciser combien d'éléments d'un certain ensemble doivent vérifier une propriété (prédicat) déterminée. On admet deux valeurs : *tous* les éléments (∀) et *au moins un* élément (∃). On pourrait en imaginer d'autres : au moins 2, exactement 5, pas plus de 8, tous sauf 1, etc. mais les logiciens ne les ont pas retenues[13].

Soit A un ensemble d'éléments et p(x) un prédicat de variable x (p peut inclure d'autres variables).

a) Quantificateur existentiel (∃)

Le quantificateur existentiel est utilisé comme suit :

$$\exists\, a \in A,\ p(a)$$

et se lit : *il existe un élément a de A tel que le prédicat p, où x est remplacé par a, est vérifié,* ou, plus simplement, *il existe dans A au moins un élément qui vérifie la condition (prédicat) P.*

L'intégrité référentielle pourrait être spécifiée à l'aide de ce quantificateur : soit c une ligne de COMMANDE. Il vient :

$$\exists\, cli \in CLIENT,\ cli.NCLI = c.NCLI$$

En toute généralité, cette propriété est vraie pour **toutes** les lignes de COMMANDE :

$$\forall\, c \in COMMANDE,\ \exists\, cli \in CLIENT,\ cli.NCLI = c.NCLI$$

Expression moins pratique assurément que la clause SQL DDL `foreign key` !

b) Quantificateur universel (∀)

Le quantificateur universel est utilisé comme suit :

$$\forall\, a \in A,\ p(a)$$

et se lit : *pour tout élément a de A, le prédicat p où x est remplacé par a, est vérifié*, ou, plus simplement, *tous les éléments de A vérifient la condition (prédicat) p*. Cette

13. Avec raison puisqu'on peut les exprimer à l'aide des constructions classiques, moyennant l'usage de la fonction |E| donnant la taille de l'ensemble E.

expression, si p ne comporte pas d'autres variables que a, a manifestement une valeur de vérité : c'est donc une proposition et non plus un prédicat.

À titre d'exemple, on pourrait exprimer le fait que la colonne DATECOM de la table COMMANDE est obligatoire par la proposition :

$\forall$ c $\in$ COMMANDE, c.DATECOM is not null

Ici encore, l'expression est moins concise que la déclaration SQL not null !

Il est possible de transformer un quantificateur dans l'autre :

$\exists x, p(x) \equiv \neg(\forall x, \neg p(x))$

$\forall x, p(x) \equiv \neg(\exists x, \neg p(x))$

La seconde règle est particulièrement importante puisque, s'il existe bien en SQL une forme correspondant au quantificateur existentiel (prédicat exists), il n'existe en revanche rien qui corresponde au quantificateur universel. C'est ce que nous allons étudier dans la section suivante.

c) Les quantificateurs en SQL

SQL propose la fonction exists (et son inverse not exists), dont l'argument est une expression renvoyant une table, et qui indique si cette table est non vide (true) ou vide (false). Cette fonction permet de simuler le quantificateur existentiel. En effet, la forme

```
select NPRO, LIBELLE
from   PRODUIT PRO
where  exists ( select *
                from   DETAIL DET
                where  NPRO = PRO.NPRO and QCOM > 10);
```

peut être interprétée comme la requête abstraite (le prédicat P encapsule la condition de la seconde clause where) :

```
select NPRO, LIBELLE
from   PRODUIT PRO
where  ∃DET ∈ DETAIL, P(DET,PRO);
```

SQL ne proposant pas de forme exprimant le quantificateur universel, nous appliquerons l'équivalence définie ci-dessus.

Recherchons par exemple *les commandes qui spécifient tous les produits*. Considérons une (ligne de) COMMANDE M. Celle-ci est sélectionnée si les PRODUITs commandés par M et les PRODUITs forment deux ensembles identiques, c'est-à-dire si le second ensemble est inclus dans le premier[14]. Ou encore, M est retenue si, *pour tout PRODUIT P, P est dans l'ensemble des PRODUITs commandés par M*. Appliquons l'équivalence

$(\forall x, p(x)) \equiv \neg(\exists x, \neg p(x))$

14. Sachant que le premier est forcément inclus dans le second.

Il vient :

la COMMANDE M est retenue s'***il n'existe pas*** *de PRODUIT P, tel que P* ***n'est pas*** *dans l'ensemble des PRODUITs commandés par M.*

La traduction mot-à-mot de cette formule en SQL ne pose pas de problèmes insurmontables :

```
la COMMANDE M est retenue      →  select NCOM from COMMANDE M
si,                            →  where
  il n'existe pas              →     not exists
    de PRODUIT P,              →       (select * from PRODUIT P
    tel que                    →         where
      P n'est pas dans         →             P.NPRO not in
        l'ensemble des PRODUITs →        (select NPRO from DETAIL
        commandés par M.       →         where NCOM = M.NCOM));
```

En rassemblant ces fragments, on obtient (l'alias P, désormais inutile, pourrait être ignoré) :

```
select NCOM
from   COMMANDE M
where  not exists (select *
                   from   PRODUIT P
                   where  P.NPRO not in (select NPRO
                                         from   DETAIL
                                         where  NCOM = M.NCOM));
```

La sous-requête interne (`select NPRO from DETAIL ...`) extrait les produits référencés par la commande courante M. La sous-requête intermédiaire (`select * from PRODUIT P ...`) définit l'ensemble des produits que M ne référence pas. La requête principale ne retient que les commandes pour lesquelles cet ensemble est vide.

Examinons deux variantes équivalentes. La première se déduit de l'observation que les valeurs de NCOM qui nous intéressent sont toutes présentes dans la table DETAIL. On peut donc réécrire la requête sous la forme :

```
select distinct NCOM
from   DETAIL M
where  not exists (select *
                   from   PRODUIT P
                   where  P.NPRO not in (select NPRO
                                         from   DETAIL
                                         where  NCOM = M.NCOM));
```

La seconde exploite la propriété qui veut que si l'ensemble A est inclus dans B, et que A et B sont de même taille, alors A = B. Considérons les lignes de DETAIL relatives à une commande. L'ensemble des valeurs de NPRO de ces lignes représente les produits commandés. S'il contient autant de valeurs qu'il y a de lignes dans la table PRODUIT, alors cette commande spécifie tous les produits. On peut alors écrire[15] :

```
select NCOM
from   DETAIL
```

```
group by NCOM
having count(distinct NPRO) = (select count(*) from PRODUIT);
```

A24.4.7 Applications pratiques en SQL

Appliquons ces règles à quelques exemples concrets, qu'on exprimera selon la syntaxe SQL.

*a) La **négation** d'une condition complexe*

Considérons d'abord les *clients de Toulouse dont le compte est négatif*. Les renseignements les concernant peuvent être obtenus par la condition composée :

```
LOCALITE = 'Toulouse' and COMPTE < 0
```

Les clients qui ne tombent pas dans cette catégorie sont caractérisés par la condition inverse, soit :

```
not (LOCALITE = 'Toulouse' and COMPTE < 0)
```

ou encore, par les lois de de Morgan (r12) :

```
not (LOCALITE = 'Toulouse') or not (COMPTE < 0)
```

ou, en simplifiant :

```
LOCALITE <> 'Toulouse' or COMPTE >= 0
```

*b) L'opérateur d'**implication***

L'opérateur d'implication ($P \Rightarrow Q$) est plus délicat à manier, en particulier parce que sa définition, telle que précisée dans la table de vérité de la section B.4.4, ne semble pas toujours conforme à l'intuition[16], qui interprète l'implication de manière plus restrictive. Comme il n'existe en SQL, nous devrons le traduire selon les opérateurs disponibles. A titre d'illustration, recherchons les clients qui, *s'ils ont un compte négatif, alors sont aussi de catégorie B1*. On peut écrire, dans un SQL *étendu* mais malheureusement fictif :

```
(COMPTE < 0) ⇒ (CAT = 'B1')
```

---

15. Dans cet exemple, le modifieur `distinct` n'est pas nécessaire. Pourquoi ?

16. Considérons la loi selon laquelle "*s'il pleut, alors la chaussée est mouillée*" ($P \equiv$ "*il pleut*" et $Q \equiv$ "*la chaussée est mouillée*"). On peut affirmer sans risque que "*il pleut et la chaussée n'est pas mouillée*" décrit une situation qui viole cette loi, d'où, en toute généralité, ($P \wedge \neg Q$) = faux. On en infère, en niant les deux membres de l'égalité, que $\neg (P \wedge \neg Q)$ = vrai. Il vient donc, par application de la règle r12, $P \Rightarrow Q \equiv \neg P \vee Q$. D'où la règle r18, qui perd ainsi tout son mystère !

L'interprétation est la suivante : si un client a un compte négatif et est de catégorie `B1`, alors il est sélectionné; bien que ceci soit moins intuitif, on admet aussi que, si son compte n'est pas négatif, alors il est sélectionné quelle que soit sa catégorie. Ou encore : les clients sont sélectionnés s'ils ont un compte non négatif ou s'ils sont de catégorie `B1` (règle r18). On peut donc réécrire la condition de sélection de ces clients en SQL pur:

```
(COMPTE >= 0) or (CAT = 'B1')
```

*c) L'opérateur **ou exclusif***

Cet opérateur stipule que l'une des conditions doit être vérifiée, et une seulement, ainsi que l'exprime la règle r19 :

$P \oplus Q \equiv$ (Q and not P) or (P and not Q)

Recherchons par exemple les clients qui n'ont pas de catégorie ou dont le compte est négatif (mais pas les deux). On peut écrire :

```
where ((CAT is null) and (COMPTE >= 0))
or    ((CAT is not null) and (COMPTE < 0))
```

ou encore, selon la règle r20 :

```
where   ((CAT is null) or (COMPTE < 0))
and not ((CAT is null) and (COMPTE < 0))
```

*d) L'opérateur d'**équivalence***

L'équivalence ($P \Leftrightarrow Q$) correspond à une double implication :$(P \Rightarrow Q) \wedge (Q \Rightarrow P)$. Cet opérateur renvoie vrai lorsque P et Q ont même valeur, soit, selon la règle r21 :

$P \Leftrightarrow Q \equiv$ (Q and P) or not (P or Q)

Recherchons les clients qui, *si leur compte est négatif, n'ont pas de catégorie*, et inversement, s'*ils n'ont pas de catégorie, alors leur compte est négatif*. Ceux qui vérifient les deux conditions, ou qui n'en vérifient aucune, sont sélectionnés. En revanche, ceux qui ne vérifient qu'une seule de ces conditions sont écartés. On écrira :

```
where  ((COMPTE < 0) and (CAT is null))
or not ((COMPTE < 0) or (CAT is null))
```

ou encore, selon la règle r13 :

```
where ((COMPTE < 0) and (CAT is null))
or     (COMPTE >= 0) and (CAT is not null) )
```

A24.4.8La logique ternaire de SQL

Bien que le langage SQL soit essentiellement basé sur la logique à deux valeurs vrai et faux[17], certains de ses aspects réclament une troisième valeur, inconnu, qui se note unknown. En effet, l'évaluation d'un prédicat, dans une clause where par exemple, sur une ligne particulière peut renvoyer la valeur vrai, la valeur faux, mais aussi la valeur inconnu, signifiant par là qu'il est impossible de décider si la valeur est vrai ou faux. Un exemple simple : la condition CAT = 'B1' renvoie la valeur de vérité unknown pour le client D063.

L'un des problèmes est que dans certaines circonstances, inconnu signifie *soit vrai soit faux*, alors qu'à d'autres moments, il signifie *ni vrai ni faux* (voir section 9.12)

Techniquement, le comportement de la valeur `inconnu` est défini par les tables de vérité des opérateurs $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\oplus$, $\Rightarrow$, et $\Leftrightarrow$ dont le contenu est donné ci-dessous[18], inc étant mis pour inconnu :

P	***Q***	$\neg$ **P**	**P** $\wedge$ **Q**	**P** $\vee$ **Q**	**P** $\oplus$ **Q**	**P** $\Rightarrow$ **Q**	**P** $\Leftrightarrow$ **Q**
vrai	***vrai***	faux	vrai	vrai	faux	vrai	vrai
vrai	***faux***	faux	faux	vrai	vrai	faux	faux
vrai	***inc***	faux	inc	vrai	inc	inc	inc
faux	***vrai***	vrai	faux	vrai	vrai	vrai	faux
faux	***faux***	vrai	faux	faux	faux	vrai	vrai
faux	***inc***	vrai	faux	inc	inc	vrai	inc
inc	***vrai***	inc	inc	vrai	inc	vrai	inc
inc	***faux***	inc	faux	inc	inc	inc	inc
inc	***inc***	inc	inc	inc	inc	inc	inc

L'élaboration de cette table est plus simple qu'il n'y paraît. La valeur de vérité de P op Q, pour chacun des opérateurs op, se détermine comme suit, lorsque P (ou Q) est inconnu : on remplace P successivement par vrai, puis par faux, et on observe les valeurs résultantes selon la table de la section B.4.4. Si les résultats de ces deux expressions sont identiques, on en inscrit la valeur dans le tableau, sinon, on inscrit inconnu.

Considérons l'expression P $\Rightarrow$ Q, lorsque P = vrai et Q = inconnu (3e ligne du tableau). Si Q = vrai, alors l'expression vaut vrai, alors qu'elle vaut faux lorsque Q = faux. Ces deux résultats étant différents, la valeur de vrai $\Rightarrow$ inconnu est donc inconnu.

17. En particulier par l'hypothèse du monde fermé, qui stipule que ce qui n'est pas vrai est faux.
18. On notera que $\neg$ inconnu = inconnu, ce qui est *logique* mais pas *intuitif*.

En SQL

Dans ce qui suit, nous utiliserons les dénominations true, false et unknown propres à SQL. Découlant de cette logique, SQL propose un jeu de prédicats binaires permettant de référencer explicitement ces trois valeurs (C est un prédicat binaire ou ternaire quelconque) :

- **C is true** vaut true si C vaut true, et false si C vaut false ou unknown;
- **C is false** vaut true si C vaut false, et false si C vaut true ou unknown;
- **C is unknown** vaut true si C vaut unknown, et false si C vaut true ou false.

Ces prédicats absorbent la valeur unknown. Les formes inverses is not true, is not false et is not unknown sont aussi disponibles.

Par exemple, la condition "(CAT = 'C2') is true" vaudra true pour le client F400, et false pour tous les autres, y compris D063 et K729.

A24.5QUE RETENIR ?

Le contenu d'une table est soit un **ensemble** soit un **multi-ensemble** de lignes. Dans le premier cas, les lignes sont distinctes et dans le second, certaines lignes peuvent être identiques. A partir d'ensembles ou de multi-ensembles, des opérateurs permettent de produire d'autres ensembles. On retient en particulier l'**union**, l'**intersection** et la **différence** d'ensembles ou de multi-ensembles. Ces opérateurs ont une traduction directe en SQL.

Plusieurs parties du langage SQL traduisent des concepts de la logique. On y retrouve les notions de **proposition** (une ligne d'une table correspond à une proposition), de **prédicat** (le schéma d'une une table est la matérialisation d'un prédicat), d'opérateurs logiques **et**, **ou** et **non**. D'autres opérateurs logiques sont utiles mais doivent être émulés en SQL. Il en est ainsi du **ou exclusif**, de l'**implication** et de l'**équivalence**. Des quantificateurs **existentiel** (∃) et **universel** (∀), SQL ne permet d'exprimer directement que le premier. L'usage du second fait appel à une traduction basée sur une équivalence. SQL n'obéit pas strictement à la logique traditionnelle, basée sur les deux valeurs de vérité **vrai** et **faux**. Il fait usage en effet d'une troisième valeur : **inconnu**.

A24.6POUR EN SAVOIR PLUS

Il existe dans le commerce de nombreux ouvrages développant les outils mathématiques utiles aux informaticiens. Sans chercher à en recommander un en particulier, nous suggérons au lecteur d'inclure [Marchand, 2005] dans leur prospection.

A24.7EXERCICES

B.1 Un amateur de bière affirme : *j'aime les brunes et les légères*. Un autre rétorque : *moi c'est les brunes et légères que je préfère*.

Reformuler ces deux assertions sous forme logique à partir des prédicats suivants :

- j'aime(X) : j'aime la bière X;
- brune(X) : la bière X est brune;
- légère(X) : la bière X est légère.

B.2 Formuler en français la négation de chacune des affirmations suivantes :

- tous les hommes sont mortels
- s'il pleut, la route est mouillée
- si tu ne manges pas tes épinards, tu n'auras pas de dessert
- tout administrateur est responsable pénalement
- il existe des informaticiens qui n'entendent rien à la logique
- toute commande comprend au moins un détail.

B.3 On tient pour vraie la proposition suivante : *s'il ne pleut pas, je n'emporte pas mon parapluie*. Quelles sont, parmi les propositions suivantes, celles qui sont également vraies ?

- si j'emporte mon parapluie, c'est qu'il pleut
- si je n'emporte pas mon parapluie, c'est qu'il ne pleut pas
- j'emmène mon parapluie seulement s'il pleut
- je n'emmène pas mon parapluie seulement s'il ne pleut pas
- il ne pleut pas seulement si je n'emporte pas mon parapluie
- il pleut seulement si j'emporte mon parapluie
- il arrive qu'il pleuve et que je n'emporte pas mon parapluie
- il arrive qu'il pleuve et que j'emporte mon parapluie
- il arrive qu'il ne pleuve pas et que je n'emporte pas mon parapluie
- il arrive qu'il ne pleuve pas et que j'emporte mon parapluie.

B.4 Que signifie, dans un langage compréhensible, l'expression courante : "vous n'êtes pas sans ignorer" ?

B.5 Ambiguïté du langage naturel ! On considère les vers suivants (P. Perret, *L'oiseau dans l'allée*) :

Papa Ronsard qu'était pas une cloche à fromage
Disait qu'il faut danser quand la musique joue

Soient les deux propositions D = "*on danse*" et M = "*la musique joue*". Comment interpréter le second vers :

- $D \Rightarrow M$.
- $M \Rightarrow D$.
- $D \Leftrightarrow M$.

Exprimer les requêtes ci-dessous en SQL.

B.6 Rechercher les clients qui, s'ils ont un compte négatif, ont passé au moins une commande. Quelle est la différence avec la requête suivante : *rechercher les clients qui ont un compte négatif et ont passé au moins une commande* ?

B.7 On considère comme ci-dessus les clients qui, s'ils ont un compte négatif, ont passé au moins une commande. Rechercher les autres clients.

B.8 Rechercher les commandes qui, si elles référencent des produits en sapin, référencent aussi des pointes en acier.

B.9 Rechercher les clients qui ont commandé le produit PA60 ou le produit PA45, mais pas les deux.

B.10 Rechercher les clients qui, s'ils ont commandé le produit PA60, en ont commandé plus de 500 unités au total.