***Date de dernière modification** : 15/7/2015*

Annexe **9**

Le langage SQL avancé

Cette annexe au chapitre 9 étudie et illustre plus particulièrement l'application des **vues SQL**, les **requêtes récursives**, la fonction agrégative **group_concat**, les **déclencheurs**, les **tables du catalogue**, l'**injection de code SQL**, les **API** d'accès aux bases de données (via l'exemple de **Python/SQLite 3**) et l'**information incomplète**.
Un jeu d'exercices est proposé en fin d'annexe.

A9.1 LES VUES SQL

On développe plus particulièrement deux techniques évoquées au chapitre 9, l'usage des vues pour mettre en oeuvre certains cas de figure d'**évolution d'un schéma** et les **vues modifiables**.

A9.1.1 Les vues comme mécanisme d'évolution de la base de données

Le script de la figure A9.1 montre les principales opérations de restructuration de la table CLIENT par *partitionnement vertical* (ou décomposition) :

- la table CLIENT est renommée XCLIENT (`alter table`)
- les deux nouvelles tables CLIENT_SIG et CLIENT_COM sont créées puis chargées des données extraites de l'ancienne table CLIENT (`create table`, `insert select`)
- une vue de nom CLIENT est créée (`create view`) sur la jointure des deux nouvelles tables.

Toutes les anciennes *requêtes de consultation* impliquant la table CLIENT sont conservées. Il n'en sera pas de même en général des *requêtes de modification* de données, pour lesquelles on sera amené à rediger des déclencheurs `instead-of`.

Cette opération de décomposition n'est cependant pas terminée. En particulier, il faudra redéfinir les objets du schéma dépendant de l'ancienne table CLIENT, notam-

ment les clés étrangères (dont NCLI de COMMANDE), les vues, les prédicats (`check`), les déclencheurs, les procédures SQL et la spécification du contrôle d'accès. L'ancienne table CLIENT, désormais dénommée XCLIENT, peut alors être archivée et/ou supprimée.

L'évolution d'une base de données est un processus inévitable mais complexe et coûteux. C'est d'ailleurs là un des arguments en faveur des bases de données dites *sans schéma* (ou *schema-less*) étudiées au chapitre 10, dans lesquelles l'accent est mis sur la souplesse d'évolution des structures, au prix bien sûr d'une absence des principaux mécanismes de contrôle de l'intégrité.

```
openDB CLICOM.db;
  alter table CLIENT rename to XCLIENT;
  create table CLIENT_SIG(
         NCLI ..,NOM ..,ADRESSE ..,LOCALITE ..);
  create table CLIENT_COM(
         NCLI ..,CAT ..,COMPTE ..);
  insert into CLIENT_SIG
         select NCLI,NOM,ADRESSE,LOCALITE from XCLIENT;
  insert into CLIENT_COM
         select NCLI,CAT,COMPTE from XCLIENT;
  create view CLIENT as
  select CS.NCLI,NOM,ADRESSE,LOCALITE,CAT,COMPTE
  from   CLIENT_SIG CS, CLIENT_COM CC
  where  CS.NCLI = CC.NCLI;
  select * from CLIENT;
closeDB;
```

Figure A9.1 - Décomposition et reconstitution de la table CLIENT

Les scripts correspondant au chapitre 9 et à cette section sont disponibles dans le fichier **Les vues SQL.sql**.

A9.1.2 Les vues modifiables

Une vue est dite *modifiable* si le SGBD est capable de traduire de manière non ambiguë les opérations de modification des données de la vue en opérations de modification des données des tables de base.

Le principe est simple : le nouvel état de la vue doit refléter toutes les modifications effectuées sur les données, et elles seulement. Pour l'utilisateur, il ne doit exister aucune différence de comportement de la base de données, que les modifications soit adressées à des données de base ou à des données virtuelles.

On a indiqué au chapitre 9 que la plupart des SGBD imposaient des contraintes très strictes sur la structure des vues modifiables, certains allant même jusqu'à les interdire (SQLite par exemple).

Heureusement (sans doute pour se faire pardonner !), la plupart des SGBD[1] autorisent le concepteur d'une vue à préciser via des **déclencheurs `instead-of`** l'effet

des opérations `insert`, `update` et `delete` sur des vues en principe non modifiables.

Considérons l'exemple de la vue DETAIL_ETENDU, définie par le script A9.2, et dont le contenu est illustré ci-dessous. Grâce à la règle d'interprétation de la jointure, nous savons que résultat compte une ligne pour chaque ligne de la table DETAIL. Son identifiant est donc (NCOM,NPRO).

NCOM	NCLI	DATECOM	NPRO	LIBELLE	PRIX	QCOM
30178	K111	2015-12-21	CS464	CHEV. SAPIN 400x6x4	220	25
30179	C400	2015-12-22	PA60	POINTE ACIER 60 (1K)	95	20
30179	C400	2015-12-22	CS262	CHEV. SAPIN 200x6x2	75	60
30182	S127	2015-12-23	PA60	POINTE ACIER 60 (1K)	95	30
30184	C400	2015-12-23	CS464	CHEV. SAPIN 400x6x4	220	120
30184	C400	2015-12-23	PA45	POINTE ACIER 45 (1K)	105	20
30185	F011	2016-01-02	PA60	POINTE ACIER 60 (1K)	95	15
30185	F011	2016-01-02	PS222	PL. SAPIN 200x20x2	185	600
30185	F011	2016-01-02	CS464	CHEV. SAPIN 400x6x4	220	260
30186	C400	2016-01-02	PA45	POINTE ACIER 45 (1K)	105	3
30188	B512	2016-01-03	PA60	POINTE ACIER 60 (1K)	95	70
30188	B512	2016-01-03	PH222	PL. HETRE 200x20x2	230	92
30188	B512	2016-01-03	CS464	CHEV. SAPIN 400x6x4	220	180
30188	B512	2016-01-03	PA45	POINTE ACIER 45 (1K)	105	22

```
create view DETAIL_ETENDU
 as
select DET.NCOM,NCLI,DATECOM,DET.NPRO,LIBELLE,PRIX,QCOM
from   COMMANDE COM,DETAIL DET,PRODUIT PRO
where  COM.NCOM = DET.NCOM
and    DET.NPRO = PRO.NPRO;
```

Figure A9.2 - Vue étendue des données de la table DETAIL

Cette vue n'est pas modifiable pour la plupart des SGBD. Or, elle devrait l'être, du moins logiquement, puisque chacune de ses lignes correspond à une unique **ligne identifiable** de COMMANDE, une **ligne identifiable** de PRODUIT et une **ligne identifiable** de DETAIL. Par exemple, la requête suivante,

```
update DETAIL_ETENDU
set    DATECOM = '2015-12-22'
where  NCOM = '30179' and NPRO = 'C400';
```

peut se traduire, sur les tables de base, par

```
update COMMANDE
set    DATECOM = '2015-12-25'
where  NCOM = '30179';
```


1. MySQL 5 reste en 2015 la principale exception.

Puisque le SGBD se déclare incapable d'interpréter de manière appropriée toutes les opérations de modification des données de la vue, c'est à nous de définir l'effet désiré des opérations `insert`, `update` et `delete`. On propose ceci :

– **`insert`** `into` DETAIL_ETENDU. Les valeurs des colonnes NCOM, NPRO et QCOM sont utilisées pour insérer une ligne dans la table de base DETAIL. Les autres valeurs sont ignorées.

– **`delete`** `from` DETAIL_ETENDU. La ligne de la table DETAIL identifiée par les valeurs de (NCOM, NPRO) est supprimée. Si cette ligne était la dernière de la ligne de COMMANDE dont elle dépendait, cette ligne de COMMMANDE est aussi supprimée, pour éviter de conserver les données de commandes sans détail.

– **`update`** DETAIL_ETENDU. Le cas de QCOM est simple : la ligne de DETAIL correspondante est modifiée. La modification de la colonne NCOM peut recevoir deux interprétations : soit modifier la valeur de l'identifiant NCOM de la ligne de COMMANDE correspondante, soit modifier la valeur de la clé étrangère NCOM de la ligne de DETAIL correspondante. Nous choisissons la seconde interprétation qui est la plus naturelle. Le raisonnement concernant NPRO est similaire. La modification des colonnes NCLI, DATECOM, LIBELLE et PRIX peut aussi donner lieu à deux interprétations. Soit cette modification est interdite via la vue DETAIL_ETENDU, soit elle est propagée vers les lignes correspondantes de COMMANDE et de PRODUIT. La colonne NCLI est **structurelle**, définissant un lien sémantique entre COMMANDE et CLIENT. Il semble raisonnable d'en interdire la modification via la vue DETAIL_ETENDU. En revanche, les colonnes DATECOM, LIBELLE et PRIX ne jouent pas de rôle important, de sorte qu'on peut en accepter la modification via cette vue.

Les scripts A9.3 et A9.4 gèrent les opérations `insert` et `delete`.

```
create trigger DE_INSERT
instead of insert on DETAIL_ETENDU
for each row
begin
  insert into DETAIL(NCOM,NPRO,QCOM)
  values(new.NCOM,new.NPRO,new.QCOM);
end;
```

Figure A9.3 - Gestion des insertions dans la vue DETAIL_ETENDU

```
create trigger DE_DELETE
instead of delete on DETAIL_ETENDU
for each row
begin
  delete from DETAIL
  where  NCOM = old.NCOM and NPRO = old.NPRO;
  delete from COMMANDE
  where  NCOM = old.NCOM
  and    NCOM not in (select NCOM from DETAIL);
end;
```

Figure A9.4 - Gestion des suppressions dans la vue DETAIL_ETENDU

Le jeu de déclencheurs gérant les modifications (`update`) des données de la vue peut prendre plusieurs formes :

- un déclencheur global :
  - `instead of update on DETAIL_ETENDU`
- un déclencheur pour chacune des trois tables de base :
  - `instead of update of NCLI,DATECOM on DETAIL_ETENDU`
  - `instead of update of NCOM,NPRO,QCOM on DETAIL_ETENDU`
  - `instead of update of LIBELLE,PRIX on DETAIL_ETENDU`
- un déclencheur pour chaque colonne modifiable de la vue.
  - `instead of update of QCOM on DETAIL_ETENDU`

C'est la troisième approche que nous développons dans le script A9.5 pour les colonnes QCOM, NCOM et LIBELLE. On notera l'opération `delete` dans le déclencheur de modification de NCOM, qui supprime la ligne de COMMANDE ancienne si elle ne possède plus de ligne de DETAIL dépendante.

Suite à l'exécution des requêtes suivantes :

```
insert into DETAIL_ETENDU(NCOM,NPRO,QCOM)
       values('30178','PA45',1111);
delete from DETAIL_ETENDU
 where  NCOM = '30182' and NPRO = 'PA60';
update DETAIL_ETENDU set QCOM = 9999
 where  NCOM = '30185';
update DETAIL_ETENDU set NCOM = '30185'
 where  NCOM = '30186';
update DETAIL_ETENDU set LIBELLE = 'indisponible'
 where  NCOM = '30185' and NPRO = 'PA60';
```

la vue DETAIL_ETENDU se présente désormais comme suit :

```
+-------+------+------------+-------+----------------------+------+------+
| NCOM  | NCLI | DATECOM    | NPRO  | LIBELLE              | PRIX | QCOM |
+-------+------+------------+-------+----------------------+------+------+
| 30178 | K111 | 2015-12-21 | CS464 | CHEV. SAPIN 400x6x4  | 220  | 25   |
| 30178 | K111 | 2015-12-21 | PA45  | POINTE ACIER 45 (1K) | 105  | 1111 |
| 30179 | C400 | 2015-12-22 | CS262 | CHEV. SAPIN 200x6x2  | 75   | 60   |
| 30179 | C400 | 2015-12-22 | PA60  | indisponible         | 95   | 20   |
| 30184 | C400 | 2015-12-23 | CS464 | CHEV. SAPIN 400x6x4  | 220  | 120  |
| 30184 | C400 | 2015-12-23 | PA45  | POINTE ACIER 45 (1K) | 105  | 20   |
| 30185 | F011 | 2016-01-02 | CS464 | CHEV. SAPIN 400x6x4  | 220  | 9999 |
| 30185 | F011 | 2016-01-02 | PA45  | POINTE ACIER 45 (1K) | 105  | 3    |
| 30185 | F011 | 2016-01-02 | PA60  | indisponible         | 95   | 9999 |
| 30185 | F011 | 2016-01-02 | PS222 | PL. SAPIN 200x20x2   | 185  | 9999 |
| 30188 | B512 | 2016-01-03 | CS464 | CHEV. SAPIN 400x6x4  | 220  | 180  |
| 30188 | B512 | 2016-01-03 | PA45  | POINTE ACIER 45 (1K) | 105  | 22   |
| 30188 | B512 | 2016-01-03 | PA60  | indisponible         | 95   | 70   |
| 30188 | B512 | 2016-01-03 | PH222 | PL. HETRE 200x20x2   | 230  | 92   |
+-------+------+------------+-------+----------------------+------+------+
```

On remarque également la disparition de la ligne de COMMANDE 30182, due à la suppression de sa dernière ligne de DETAIL :

```
+-------+------+------------+
| NCOM  | NCLI | DATECOM    |
+-------+------+------------+
| 30178 | K111 | 2015-12-21 |
| 30179 | C400 | 2015-12-22 |
| 30184 | C400 | 2015-12-23 |
| 30185 | F011 | 2016-01-02 |
| 30188 | B512 | 2016-01-03 |
+-------+------+------------+
```

```
create trigger DE_UPDATE_QCOM
instead of update of QCOM on DETAIL_ETENDU
for each row
begin
  update DETAIL
  set    QCOM = new.QCOM
  where  NCOM = old.NCOM
  and    NPRO = old.NPRO;
end;
create trigger DE_UPDATE_NCOM
instead of update of NCOM on DETAIL_ETENDU
for each row
begin
  update DETAIL
  set    NCOM = new.NCOM
  where  NCOM = old.NCOM
  and    NPRO = old.NPRO;
  delete from COMMANDE
  where  NCOM = old.NCOM
  and    NCOM not in (select NCOM from DETAIL);
end;
create trigger DE_UPDATE_LIBELLE
instead of update of LIBELLE on DETAIL_ETENDU
for each row
begin
  update PRODUIT set LIBELLE = new.LIBELLE
  where  NPRO = old.NPRO;
end;
```

Figure A9.5 - Déclencheurs de gestion des modifications de QCOM, NCOM et LIBELLE dans la vue DETAIL_ETENDU

L'écriture des déclencheurs peut s'avérer complexe et fastidieuse pour un grand nombre de vues modifiables. On pensera alors à automatiser leur génération à partir des tables du catalogue. La définition des vues se fera via une interface graphique également pilotée par les tables du catalogue.

Les scripts correspondant à cette section sont disponibles dans le fichier **Les vues SQL modifiables.sql**.

Le section elle-même a été traduite sous la forme d'un tutoriel : **9._Les_vues_SQL_modifiables.tuto**.

A9.1.3 Vues modifiables et O/RM

L'usage de vues complexes modifiables est l'un des principaux objectifs des O/RM (Object/Relational Mapping), ces logiciels intermédiaires (*middleware*) qui offrent au code applicatif une vue objet des données d'une base de données SQL.

La technique des vues modifiables peut dans certains cas, se substituer à l'utilisation d'un O/RM. L'avantage des vues modifiables sur ce dernier est triple :

- La définition et la gestion des transformations tant en consultation qu'en mise à jour sont **centralisées** au niveau de la base de données et de son serveur.
- Toute la puissance du langage SQL reste applicable aux vues modifiables.
- La conversion vue/objet devient triviale puisque la structure des deux est identique, et ne nécessite plus qu'un mapping *un-à-un*.

A9.2 LES REQUÊTES RÉCURSIVES

Nous proposons dans cette section une étude plus approfondie et quelques applications utiles des requêtes SQL récursives. Nous nous intéresserons plus particulièrement au traitement des structures de graphes à partir des deux exemples PERSONNE et (PRODUIT, COMPOSITION) décrits au chapitre 8. Les scripts développés sont rassemblés dans deux fichiers : **Les requetes recursives (PERSONNE).sql** et **Les requetes recursives (COMPOSITION).sql**.

A9.2.1 Chemins dans un arbre (PERSONNE)

Reprenons l'exemple de la table PERSONNE du chapitre 8, à laquelle nous apportons une légère modification : la personne p2 est désormais un subordonné de p1 (figure A9.6).

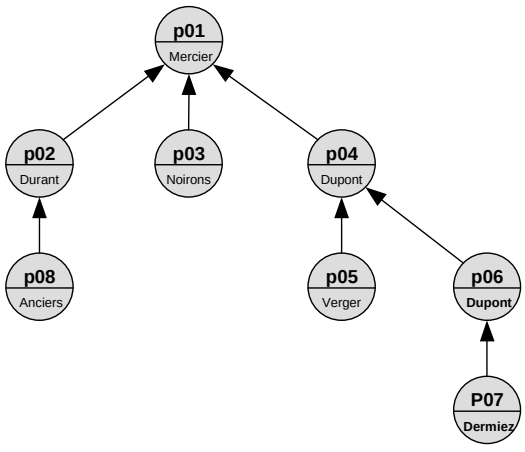

Figure A9.6 - Nouvelle version de la structure des données de la table PERSONNE

La requête A9.7 construit tous les chemins existant dans la table PERSONNE. On admet que chaque personne constitue à elle seule un chemin élémentaire de longueur 0. Le plus long chemin est celui qui part de p1 et qui aboutit à p7. Il est de longueur 3.

```
with recursive ORG(Premier,Dernier,Longueur,Chemin)
as (select NPERS,NPERS,0,'.'||NPERS||'.' from PERSONNE
      union all
    select O.Premier,P.NPERS,O.Longueur + 1,
           O.Chemin||P.NPERS||'.'
    from   ORG O, PERSONNE P
    where  O.Dernier = P.RESPONSABLE)
    select * from ORG order by Chemin;
```

Figure A9.7 - Génération des chemins de la table PERSONNE

Premier	Dernier	Longueur	Chemin
p1	p1	0	.p1.
p1	p2	1	.p1.p2.
p1	p8	2	.p1.p2.p8.
p1	p3	1	.p1.p3.
p1	p4	1	.p1.p4.
p1	p5	2	.p1.p4.p5.
p1	p6	2	.p1.p4.p6.
p1	p7	3	.p1.p4.p6.p7.
p2	p2	0	.p2.
p2	p8	1	.p2.p8.
p3	p3	0	.p3.
p4	p4	0	.p4.
p4	p5	1	.p4.p5.
p4	p6	1	.p4.p6.
p4	p7	2	.p4.p6.p7.
p5	p5	0	.p5.
p6	p6	0	.p6.
p6	p7	1	.p6.p7.
p7	p7	0	.p7.
p8	p8	0	.p8.

A9.2.2 Les graphes non acycliques

La structure du contenu de la table PERSONNE est un **arbre**, ou plus généralement, si on admet plus d'un directeur, une **forêt**. Que se passerait-il si nous déclarions, par erreur ou provocation, que **p7 est le responsable de p1** ?

```
update PERSONNE set RESPONSABLE = 'p7' where NPERS = 'p1'
```

La réponse est simple : la requête boucle[2] ! En effet, le graphe contient désormais un circuit (figure A9.8). Partant d'un des noeuds de ce circuit, la requête calcule de

manière récursive l'ensemble de ses subordonnés, réexaminant sans fin ce noeud initial.

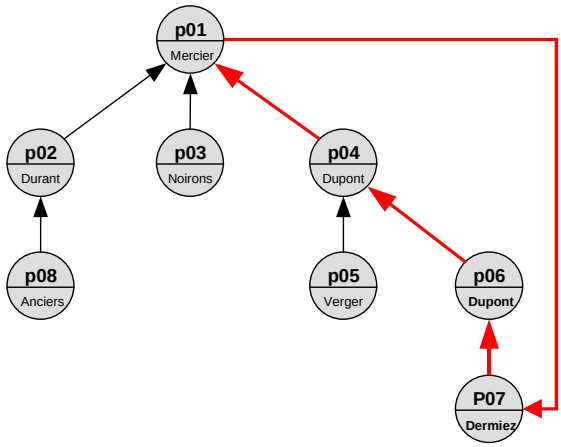

Figure A9.8 - Le graphe de PERSONNE comporte un circuit

Cet *incident*[3] nous suggère un problème intéressant : comment éviter ce bouclage infini ? Le SGBD lui-même ne nous apportera pas d'aide très utile. Si certains proposent d'associer à la requête un paramètre limitant le nombre d'appels récursifs à une constante prédéfinie (SQL Server par exemple), d'autres n'imposent aucune limite (SQLite par exemple). La solution viendra de la forme de la requête elle-même. On propose trois techniques.

1. Limiter le nombre d'appels récursifs à une **constante arbitraire**. On admet qu'une hiérarchie de profondeur supérieure à 10 est irréaliste dans une organisation bien gérée. On arrêtera donc les appels dès que le niveau 10 est atteint (script A9.9).
2. Limiter le nombre d'appels récursifs à la taille du **cas le plus défavorable**. Le cas limite d'une hiérarchie est celui d'une simple chaîne : il y a un seul directeur et chaque responsable supervise une seule personne. Le niveau maximum est dans ce cas égal au nombre de personnes. C'est cette limite naturelle qu'on adopte dans le script A9.10.
3. **Traiter chaque noeud une fois au plus**. On arrête les appels récursifs dès qu'on rencontre une personne déjà examinée. Cette condition est facile à évaluer par l'examen du chemin courant : on vérifie que la valeur courante de NPERS n'est pas présente dans la valeur courante de Chemin (par l'opérateur

2. Dans le cas de la version de base de SQLfast, il n'y a pas d'autre solution que de tuer le processus Python. Le bouclage est en effet entièrement sous le contrôle (ou absence de contrôle) de SQLite.
3. L'existence d'un circuit n'est pas en soi un incident ni un défaut. Il l'est ici car il viole la structure d'arbres que la table est sensée représenter.

`like`). Cette solution est la plus élégante car elle construit un résultat pertinent, contrairement aux deux premières (script A9.11). Elle est d'ailleurs généralisable à plus d'une seule occurrence des noeuds visités : traiter chaque noeud 2, 3, ... fois au plus.

```
with recursive ORG(Niveau,NPERS,Chemin)
as (select 1,'$pers$','.$pers$.'
      union all
    select Niveau+1,P.NPERS,Chemin||P.NPERSD||'.'
    from   ORG O, PERSONNE P
    where  O.NPERS = P.RESPONSABLE
    and    Niveau < 10
    )
select * from ORG order by Chemin;
```

Figure A9.9 - Contrôle de la récursion : on impose une limite arbitraire

```
extract N = select count(*) from PERSONNE;
with recursive ORG(Niveau,NPERS,Chemin)
as (select 1,'$pers$','.$pers$.'
      union all
    select Niveau+1,P.NPERS,Chemin||P.NPERSD||'.'
    from   ORG O, PERSONNE P
    where  O.NPERS = P.RESPONSABLE
    and    Niveau < $N$
    )
select * from ORG order by Chemin;
```

Figure A9.10 - Contrôle de la récursion : par le cas le plus défavorable

```
with recursive ORG(Niveau,NPERS,Chemin)
as (select 1,'$pers$','.$pers$.'
      union all
    select Niveau+1,P.NPERS,Chemin||P.NPERSD||'.'
    from   ORG O, PERSONNE P
    where  O.NPERS = P.RESPONSABLE
    and    Chemin not like '%.'||P.NPERS||'.%'
    )
select * from ORG order by Chemin;
```

Figure A9.11 - Contrôle de la récursion : chaque noeud est examiné une seule fois

Les scripts de cette section sont disponibles dans le fichier **Les requetes recursives (PERSONNE).sql**.

A9.2.3 Les graphes hiérarchiques (PRODUIT, COMPOSITION)

Alors que dans un graphe arborescent, comme celui des données de la table PERSONNE, un noeud possède au plus un noeud parent (ici le *responsable*), un noeud d'un graphe hiérarchique peut dépendre de plusieurs parents. Tel est le cas des données des tables PRODUIT et COMPOSITION étudiées au chapitre 8, dont on reprend la représentation graphique à la figure A9.12.

Nous allons extraire de ces données tous les triplets de composition (P,C,Q) tels que la quantité Q du produit C entre, directement ou indirectement, dans la composition d'une unité du produit P.[4] Le calcul de ces triplets est effectué par la requête de la figure A9.13.

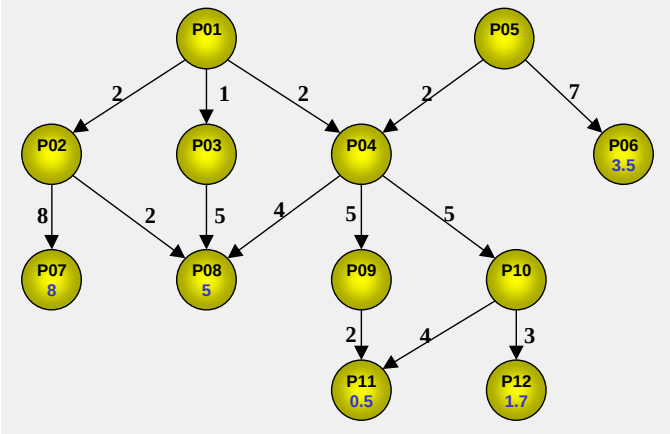

Figure A9.12 - Graphe des données des tables PRODUIT et COMPOSITION

```
with recursive CHAINE(Produit,Compo,Qte) as
    (select NPRO,NPRO,1 from PRODUIT
       union all
     select Ch.Produit,Co.COMPOSANT,Ch.Qte * Co.QTE
     from   CHAINE Ch, COMPOSITION Co
     where  Ch.Compo = Co.COMPOSE
     )
select * from CHAINE
where Produit <> Compo order by Produit,Compo;
```

Figure A9.13 - Génération de toutes les chaînes *composé-composant*

La requête d'initialisation introduit dans la vue l'ensemble des produits, associés chacun à une quantité unitaire de lui-même (pour obtenir une unité du produit P, il faut ... une unité de P !). La requête récursive ajoute à la vue la composition (au sens

4. L'ensemble des couples (P,C) ainsi produits constitue la *fermeture transitive* du graphe.

mathématique du terme) des triplets existants non encore traités avec les données de la table COMPOSITION. La requête principale extrait ces triplets, en excluant toutefois ceux de la requête d'initialisation, désormais sans intérêt. Le résultat est le suivant :

Chaînes de composition

Produit	**Compo**	**Qte**
p01	p02	2
p01	p03	1
p01	p04	2
p01	p07	16
p01	**p08**	4
p01	**p08**	5
p01	**p08**	8
p01	p09	10
p01	p10	10
p01	p11	20
p01	p11	40
p01	p12	30
p02	p07	8
p02	p08	2
p03	p08	5
p04	p08	4
p04	p09	5
p04	p10	5
p04	p11	10
p04	p11	20
p04	p12	15
p05	p04	2
p05	p06	7
p05	p08	8
p05	p09	10
p05	p10	10
p05	p11	20
p05	p11	40
p05	p12	30
p09	p11	2
p10	p11	4
p10	p12	3

On observe que certains couples (P,C) existent en plusieurs exemplaires, signalant ainsi que le produit C est un composant de P selon plusieurs chemin différents. Tel est le cas par exemple de (p01,p08) présent en trois exemplaires (en gras dans le tableau ci-dessus).

À partir de ce résultat, on calcule aisément la consommation de chaque produit composé en matières premières par la requête principale du script A9.14, lequel est suivi du résultat de son exécution.

Pour simplifier la rédaction des requêtes, on a au préalable défini deux vues SQL, MATIERE_PREMIERE et PRODUIT_FINI, dont la définition et la signification sont supposées évidentes.

```
with recursive CHAINE(Produit,Compo,Qte) as
    (idem)
select Produit,Compo as "Mat. Premiere",sum(QTE) as Quantite
from   CHAINE
where  Compo in (select NPRO from MATIERE_PREMIERE)
and    Produit <> Compo
group by Produit,Compo
order by Produit,Compo;
```

Figure A9.14 - Calcul de la consommation des produits composés en matières premières

Consommation des produits composés en matières premières

Produit	**Mat. Premiere**	**Quantite**
p01	p07	16
p01	p08	17
p01	p11	60
p01	p12	30
p02	p07	8
p02	p08	2
p03	p08	5
p04	p08	4
p04	p11	30
p04	p12	15
p05	p06	7
p05	p08	8
p05	p11	60
p05	p12	30
p09	p11	2
p10	p11	4
p10	p12	3

En limitant la sortie aux **produits finis** (ici p01 et p05) et en valuant les quantités selon le prix unitaire correspondant, on obtient le **coût** en **matières premières** des **produits finis** (script A9.15).

```
with recursive CHAINE(Produit,Compo,Qte) as
    (idem)
select Produit,sum(QTE*PRIX_U) as Cout
from   CHAINE Ch, PRODUIT Pro
where  Produit in (select NPRO from PRODUIT_FINI)
and    Compo   in (select NPRO from MATIERE_PREMIERE)
and    Compo = Pro.NPRO
and    Produit <> Compo
group by Produit;
```

Figure A9.15 - Calcul du coût en matières premières des produits finis

Coût unitaire en matières premières des produits finis

Produit	Cout
p01	294.0
p05	145.5

A9.2.4 Mise à jour du prix des produits composés

Si toutes les matières premières ont reçu un coût unitaire, alors il est facile de calculer le coût unitaire des produits composés par un update s'appuyant sur la vue récursive CHAINE. C'est ce que réalise le script A9.16.

```
extract N = select count(*)
            from   MATIERE_PREMIERE
            where  PRIX_U is null;

if ($N$ > 0);
  write Données des matières premières incomplètes.;
  write La table PRODUIT ne peut être mise à jour.;
else;
  with recursive CHAINE(Produit,Compo,Qte) as
      (idem)
  update PRODUIT
  set    PRIX_U = (select sum(Qte*PRIX_U)
                   from   CHAINE, PRODUIT Pr
                   where  Compo = Pr.NPRO
                   and    Compo in (select NPRO
                                    from   MATIERE_PREMIERE)
                   and    Produit = PRODUIT.NPRO
                   group by Produit)
  where  NPRO in (select COMPOSE from COMPOSITION);
endif;
```

Figure A9.16 - Mise à jour du prix unitaire des produits composés

NPRO	LIBELLE	PRIX_U
p01	A-200	294
p02	A-056	74
p03	B-661	25
p04	B-122	60.5
p05	B-326	145.5
p06	D-822	3.5
p07	D-507	8
p08	G-993	5
p09	F-016	1
p10	J-500	7.1
p11	J-544	0.5
p12	L-009	1.7

A9.3 LA FONCTION AGRÉGATIVE `group_concat`

La fonction `group_concat` (ou son équivalent selon le SGBD utilisé) permet d'éviter le recours à la programmation procédurale dans de nombreuses applications. Nous proposerons comme exemples des extractions simples de la table CLIENT puis nous expérimenterons une utilisation à deux niveau de cette fonction.

Les requêtes seront rédigées dans la version de base de SQLfast qui inclut SQLite. Rappelons que dans cette variante de SQL, la fonction `group_concat` est disponible sous une forme simplifiée n'utilisant que deux arguments :

```
group_concat(<expression>,<séparateur>)
```

Les scripts de cette section sont disponibles dans le fichier **group_concat.sql**.

A9.3.1 Extractions de données agrégées de la table CLIENT

La requête A9.17 extrait, pour chaque produit, la liste les localités dans lesquelles ce produit est commandé.

```
select P.NPRO as Produit,
       group_concat(LOCALITE,',') as Localités
from   CLIENT C,COMMANDE M,DETAIL D,PRODUIT P
where  C.NCLI = M.NCLI and M.NCOM = D.NCOM and D.NPRO = P.NPRO
group by P.NPRO;
```

Figure A9.17 - Une première tentative d'extraire les localités dans lesquelles chaque produit est commandé

```
+---------+---------------------------------+
| Produit | Localités                       |
+---------+---------------------------------+
| CS262   | Poitiers                        |
| CS464   | Lille,Poitiers,Toulouse,Toulouse |
| PA45    | Poitiers,Poitiers,Toulouse      |
| PA60    | Poitiers,Namur,Toulouse,Toulouse |
| PH222   | Toulouse                        |
| PS222   | Toulouse                        |
+---------+---------------------------------+
```

Le résultat obtenu n'est pas tout à fait satisfaisant : des doublons apparaissent dans la liste des localités et cette liste n'est pas ordonnée. En MySQL, le résultat désiré serait obtenu par la fonction suivante, grâce aux paramètres `distinct` et `order by` :

```
group_concat(distinct LOCALITE order by LOCALITE asc
             separator ',')
```

En SQLite, qui ne dispose pas de ces paramètres, on imposera l'unicité et l'ordre dans la clause `from` de la requête principale (script A9.18).

```
select  Produit,group_concat(Localite,',') as Localites
from   (select distinct P.NPRO as Produit,LOCALITE as Localite
        from   CLIENT C,COMMANDE M,DETAIL D,PRODUIT P
        where  C.NCLI = M.NCLI and M.NCOM = D.NCOM
        and    D.NPRO = P.NPRO
        order by Produit,Localite)
group by Produit;
```

Figure A9.18 - Extraction des localités dans lesquelles chaque produit est commandé

```
+---------+-------------------------+
| Produit | Localites               |
+---------+-------------------------+
| CS262   | Poitiers                |
| CS464   | Lille,Poitiers,Toulouse |
| PA45    | Poitiers,Toulouse       |
| PA60    | Namur,Poitiers,Toulouse |
| PH222   | Toulouse                |
| PS222   | Toulouse                |
+---------+-------------------------+
```

A9.3.2 Agrégation multi-niveaux

Le résultat d'une requête utilisant une fonction `group_concat` peut lui-même faire l'objet d'une fonction `group_concat`. C'est ce que montre le script A9.19 qui présente une version synthétique des commandes des clients.

```
select Client,group_concat(Commande,';') as Commandes
from (select NCLI as Client,
             M.NCOM||'['||group_concat(NPRO,',')||']' as Commande
      from   COMMANDE M,DETAIL D
      where  M.NCOM = D.NCOM
      group by NCLI,M.NCOM)
group by Client;
```

Figure A9.19 - Synthèse des commandes par client

```
+--------+-----------------------------------------------+
| Client | Commandes                                     |
+--------+-----------------------------------------------+
| B512   | 30188[CS464,PA45,PA60,PH222]                  |
| C400   | 30179[CS262,PA60];30184[CS464,PA45];30186[PA45] |
| F011   | 30185[CS464,PA60,PS222]                       |
| K111   | 30178[CS464]                                  |
| S127   | 30182[PA60]                                   |
+--------+-----------------------------------------------+
```

A9.3.3 Génération d'objets complexes

La fonction `group_concat` est l'outil idéal pour transformer les données relationnelles en objets complexes. En modifiant légèrement la dernière requête, on obtient un script générant des objets JSON (*Javascript Object Notation*).

```
select '{"NCLI":"'||Client||'","Commandes":['
     ||group_concat(Commande,',')||']}' as JSON
from  (select NCLI as Client,
             '{"NCOM":"'||M.NCOM||'","Details":['
           ||group_concat('"'||NPRO||'"',',')||']}' as Commande
      from   COMMANDE M,DETAIL D
      where  M.NCOM = D.NCOM
      group by NCLI,M.NCOM)
group by Client;
```

Figure A9.20 - Génération d'objets JSON

L'objet JSON relatif au client C400 se présente comme suit :

```
{"NCLI":"C400",
 "Commandes":[{"NCOM":"30179","Details":["CS262","PA60"]},
              {"NCOM":"30184","Details":["CS464","PA45"]},
              {"NCOM":"30186","Details":["PA45"]}
             ]
}
```

Le même principe est applicable à la génération d'autres formats d'échange tels que XML ou CSV.

A9.4 LES DÉCLENCHEURS

Nous examinons dans cette section quelques applications typiques et avancées des déclencheurs

A9.4.1 Nature du corps d'un déclencheur

La partie *Action* d'un déclencheur est une suite d'actions analogue au corps d'une procédure SQL (*Stored Procedure*). Les standards en cette matière sont très restrictifs : le corps d'une procédure est une *séquence de requêtes SQL*. Il n'inclut donc pas de structures de contrôles (alternatives, choix multiples, boucles, appel d'autres procédures, etc.), de variables ou d'appel à des services extérieurs. Tout au plus peut-on générer une exception en cas d'*incident*. Comme on pouvait s'y attendre, la plupart des SGBD ont dépassé cette vision limitée en proposant un langage procédural plus ou moins complet, soit propriétaire (par exemple *PL/SQL* chez Oracle et *Transact-SQL* chez SQL Server), soit généraux (Java par exemple).

Dans ce contexte, la position de **SQLite** est intéressante. Le corps d'un déclencheur SQLite est strictement limité à **une séquence de requêtes SQL**. Cependant,

l'absence de construction alternative (*if-then-else)* et de choix multiple (*case*) peut être compensée de plusieurs manières:

- le déclencheur SQLite dispose d'une clause **when** qui autorise des conditions SQL de complexité quelconque, pouvant impliquer plusieurs tables et des fonctions agrégatives.
- SQLite autorise la définition de **plusieurs déclencheurs** sur la même table et sur les mêmes événements. Il est ainsi possible de simuler une structure de choix multiples.
- Une condition peut être ajoutée à toute requête SQL de manière à en définir l'exécution conditionnelle. Couplée à l'usage de la clause **when**, ces conditions SQL permettent de construire naturellement deux niveaux de décision.

Par exemple, le trigger suivant, rédigé dans un pseudo-code quelconque :

```
create trigger TRG_TEST
before delete on T1
for each row
begin
  if (<C1>)
  then delete from T2 where <C2>;
  else delete from T3 where <C3>;
  endif;
end;
```

peut aussi s'écrire :

```
create trigger TRG_TEST
before delete on T1
for each row
begin
  delete from T2 where <C1> and <C2>;
  delete from T3 where not <C1> and <C3>;
end;
```

ou encore, sous la forme de deux déclencheurs complémentaires :

```
create trigger TRG_TEST1
before delete on T1
for each row
when <C1>
begin
  delete from T2 where <C2>;
end;

create trigger TRG_TEST2
before delete on T1
for each row
when not <C1>
begin
  delete from T3 where <C3>;
end;
```

L'annulation des opérations s'effectue classiquement par la génération d'une exception. Celle-ci s'accompagne concrètement de l'émission d'informations à destina-

tion du programme client précisant la nature et les causes de l'incident. Dans un déclencheur SQLite cette action prend la forme d'une variante de la requête **`select`** dont les éléments de la liste `select` se réduisent à la fonction `raise`. Par exemple,

```
create trigger TRG_TEST1
before insert on CLIENT
for each row
when not (new.CAT is null or CAT in ('B1','B2','C1','C2'))
begin
  select raise(ABORT,'Valeur de CAT invalide');
end;
```

ou encore

```
create trigger TRG_TEST1
before insert on CLIENT
for each row
begin
  select raise(ABORT,'Valeur de CAT invalide')
  where  not (new.CAT is null
              or CAT in ('B1','B2','C1','C2'));
end;
```

Le corps du déclencheur peut comporter plusieurs générateurs d'exception. En SQLfast, l'exception assigne à la variable système **SQLdiag** la valeur `'OTHER'` et à la variable **EXTENDEDdiag** le message de la fonction **`raise`**. Des exemples d'application sont proposés dans le fichier **Declencheurs et exceptions.sql**.

A9.4.2 Gestion de contraintes d'intégrité

La gestion des contraintes d'intégrité à l'aide d'un déclencheur est assez simple, puisque son action est le plus généralement d'annuler l'effet de l'opération de modification litigieuse. Soit <Condition> la traduction en SQL d'une contrainte d'intégrité. L'expression du déclencheur destiné à garantir la satisfaction en temps réel de cette contrainte peut prendre différentes formes. Nous en donnerons ci-après deux variantes selon la syntaxe générique utilisée dans l'ouvrage et deux autres selon la syntaxe SQLite.

Expressions générique	Expressions en SQLite

create ...
when not <condition>
begin
  raise <parametres>;
end;
	create ...
	when not <condition>
	begin
	  select raise(ABORT,<message>;
	end;

create ...
when not <condition>
begin
  if (not <condition>)
  then raise <parametres>;
  endif;
end;
	create ...
	when not <condition>
	begin
	  select raise(ABORT,<message>)
	  where not <condition>;
	end;

A9.4.3 Gestion de la redondance

Le problème

Pour illustrer le rôle des déclencheurs dans la gestion de la redondance, nous allons enrichir le schéma CLICOM de trois colonnes calculées : MONTANT, TOTAL_COM et TOTAL_CLI. En outre, ces colonnes dépendent l'une de l'autre comme suit :

– La colonne **MONTANT** de la table DETAIL a pour valeur le produit de la colonne QCOM et de la colonne PRIX de la ligne PRODUIT référencée. De manière plus précise, on définit MONTANT comme suit :

```
∀ det ∈ DETAIL,
        det.MONTANT = det.QCOM * (select PRIX
                                  from   PRODUIT
                                  where  NPRO = det.NPRO)
```

De l'examen de cette relation, on conclut que la valeur de MONTANT doit être (re)calculée après les quatre opérations suivantes : insertion dans DETAIL, modification de DETAIL.QCOM, modification de PRODUIT.PRIX et modification de DETAIL.NPRO. La modification de PRODUIT.NPRO est automatiquement gérée via la clé étrangère (*on update cascade*). La suppression de PRODUIT est sans effet, toujours en raison de la clé étrangère (*on update no action*).

– La colonne **TOTAL_COM** de la table COMMANDE a pour valeur pour chaque ligne, la somme des valeurs de MONTANT des lignes de DETAIL référençant cette ligne. Soit encore, de manière plus précise :

```
∀ com ∈ COMMANDE,
        com.TOTAL_COM = select sum(MONTANT)
                        from   DETAIL
                        where  NCOM = com.NCOM
```

On convient que l'expression de droite vaut 0 si aucune ligne de DETAIL n'est attachée à la ligne com de COMMANDE.
On conclut que la valeur de TOTAL_COM doit être (re)calculée après les cinq opérations suivantes : insertion dans COMMANDE, insertion dans DETAIL, suppression dans DETAIL, modification de DETAIL.MONTANT et modification de DETAIL.NCOM. La modification de COMMANDE.NCOM est automatiquement gérée via la clé étrangère (*on update cascade*).

– Enfin, la colonne **TOTAL_CLI** de la table CLIENT reprend, pour chaque ligne, la somme des valeurs de TOTAL_COM des lignes de COMMANDE référençant cette ligne. Son expression est similaire à celle de TOTAL_COM :

```
∀ cli ∈ CLIENT,
        cli.TOTAL_CLI = select sum(TOTAL_COM)
                        from   COMMANDE
                        where  NCLI = cli.NCLI
```

On convient que l'expression de droite vaut 0 si aucune ligne de COMMANDE n'est attachée à la ligne cli de CLIENT.

La valeur de TOTAL_CLI doit être (re)calculée après les quatre opérations suivantes : insertion dans CLIENT, suppression dans COMMANDE, modification de COMMANDE.TOTAL_COM et modification de COMMANDE.NCLI. La modification de CLIENT.NCLI est automatiquement gérée via la clé étrangère. Remarquons que l'insertion dans COMMANDE n'a pas d'effet sur TOTAL_CLI puisque, après cette opération, TOTAL_COM = 0.

Analyse du problème

A. Événements affectant la table DETAIL

Recherchons les événements relatifs à la table DETAIL susceptibles d'entraîner indirectement une modification des colonnes calculées. Définissons pour chacun la réaction appropriée.

Événement	Réaction
insert into **DETAIL**	initialiser MONTANT à 0; calculer la valeur de MONTANT de la **nouvelle** ligne de DETAIL
delete from **DETAIL**	actualiser la valeur de TOTAL_COM de la ligne de COMMANDE référencée
update **DETAIL** set NPRO	actualiser la valeur de MONTANT;
update **DETAIL** set NCOM	actualiser la valeur de TOTAL_COM de la **nouvelle** ligne de COMMANDE référencée; actualiser la valeur de TOTAL_COM de l'**ancienne** ligne de COMMANDE référencée
update **DETAIL** set QCOM	actualiser la valeur de MONTANT
update **DETAIL** set MONTANT	calculer la valeur de TOTAL_COM de la ligne de COMMANDE référencée

B. Événements affectant la table PRODUIT

Même raisonnement pour la table PRODUIT

Événement	Réaction
update **PRODUIT** set PRIX	recalculer la valeur de MONTANT de toutes les lignes de DETAIL dépendante

C. Événements affectant la table COMMANDE

Événement	Réaction
insert into **COMMANDE**	assigner 0 à la colonne TOTAL_COM

delete from **COMMANDE**	actualiser la valeur de TOTAL_CLI de la ligne de CLIENT référencée
update **COMMANDE** set NCLI	actualiser la valeur de TOTAL_CLI de la **nouvelle** ligne de CLIENT référencée; actualiser la valeur de TOTAL_CLI de l'**ancienne** ligne de CLIENT référencée;
update **COMMANDE** set TOTAL_COM	actualiser la valeur de TOTAL_CLI de la ligne de CLIENT référencée;

D. Événements affectant la table CLIENT

Événement	Réaction
insert into **CLIENT**	assigner 0 à la colonne TOTAL_CLI

Solution

Dans le cas de l'`insert` de CLIENT, de COMMANDE et de DETAIL, la spécification d'une valeur par défaut `0` suffira. Pour chacun des autres événements, nous rédigerons un déclencheur spécifique.

Les colonnes calculées ne doivent pas être modifiables directement par l'utilisateur (un programme client par exemple). On réglera ce problème via le contrôle des accès :

```
revoke update (TOTAL_CLI) on CLIENT   from public;
revoke update (TOTAL_COM) on COMMANDE from public;
revoke update (MONTANTI)  on DETAIL   from public;
```

Pas d'inquiétude, ces restrictions ne s'appliquent pas aux déclencheurs, même s'ils sont activés par un utilisateur qui n'a pas l'autorisation d'`update` sur ces colonnes. Il sera donc possible à un déclencheur de modifier la valeur des colonnes calculées.

L'actualisation de TOTAL_COM et TOTAL_CLI peut se faire de deux manières :

1. Par recalcul de leur valeur selon la formule spécifiée ci-dessus. C'est la technique la plus simple mais qui peut s'avérer coûteuse si le nombre de lignes dépendantes est important. C'est cette technique que nous illustrerons.
2. Par ajustement de leur valeur actuelle en retirant, pour la ligne dépendante concernée, la valeur ancienne et/ou en ajoutant la valeur nouvelle.

Les figures A9.21 à A9.23 donnent le code de trois déclencheurs représentatifs. On observera l'usage de la fonction `coalesce`, qui évite l'assignation de la valeur null à la colonne TOTAL_COM si l'ancienne ligne de COMMANDE perd sa dernière ligne de DETAIL dépendante.

```
create trigger TRG_DET_INS
after insert on DETAIL
for each row
begin
  update DETAIL
  set    MONTANT = new.QCOM * (select PRIX
                               from   PRODUIT
                               where  NPRO = new.NPRO)
  where  NCOM = new.NCOM and NPRO = new.NPRO;
end;
```

Figure A9.21 - Déclencheur calculant la valeur de MONTANT d'une nouvelle ligne de DETAIL

```
create trigger TRG_DET_UPD_NCOM
after update of NCOM on DETAIL
for each row
begin
  update COMMANDE
  set    TOTAL_COM = coalesce((select sum(MONTANT)
                               from   DETAIL
                               where  NCOM = old.NCOM),0)
  where  NCOM = old.NCOM;
  update COMMANDE
  set    TOTAL_COM = (select sum(MONTANT)
                      from   DETAIL
                      where  NCOM = new.NCOM)
  where  NCOM = new.NCOM;
end;
```

Figure A9.22 - Déclencheur actualisant les valeurs de TOTAL_COM suite à la modification de **NPRO** de DETAIL

```
create trigger TRG_DET_UPD_MONTANT
after update of MONTANT on DETAIL
for each row
begin
  update COMMANDE
  set    TOTAL_COM = (select sum(MONTANT)
                      from   DETAIL
                      where  NCOM = new.NCOM)
  where  NCOM = new.NCOM;
end;
```

Figure A9.23 - Déclencheur actualisant la valeur de TOTAL_COM suite à la modification de **MONTANT** de DETAIL

Ordre de déclenchement

Les déclencheurs sont conçus pour gérer (correctement) les cas d'événements élémentaires, tels que la modification d'une seule colonne. Mais en ira-t-il de même des modifications multiples telles que demandées dans la requête suivante :

```
update DETAIL
set NCOM = '30186',
    NPRO = 'PA60',
    QCOM = 68
where NCOM = '30185' and NPRO = 'PS222';
```

Dans un tel cas en effet, trois déclencheurs seront activés, chacun pour une des modifications élémentaires de la requête. La question critique est celle de leur ordre de déclenchement. Les valeurs **finales** de MONTANT, de TOTAL_COM et de TOTAL_CLI dépendent-elles de cet ordre ?

L'analyse du corps des déclencheurs montre que la valeur de MONTANT est calculée à partir des valeurs de new.QCOM et de new.NPRO (qui conduit à la nouvelle valeur de PRIX) de l'état final de la ligne en modification. La valeur de MONTANT sera recalculée deux fois, mais, quel que soit l'ordre de déclenchement de TRG_DET_UPD_QCOM (update de QCOM) et TRG_DET_UPD_NPRO (update de NPRO), la valeur finale de MONTANT sera la même.

Examinons à présent l'actualisation de TOTAL_COM.

– Supposons que la valeur de NCOM soit réalisée **en premier**. L'ancienne ligne de COMMANDE référençée est actualisée par soustraction de l'ancienne valeur de MONTANT, ce qui est correct, puis la nouvelle ligne est actualisée, par addition de l'ancienne valeur de MONTANT, ce qui est prématuré. Ensuite, la valeur de MONTANT est modifiée, ce qui entraîne l'actualisation (en deux étapes) de la nouvelle ligne de COMMANDE. Au final, les valeurs de TOTAL_COM des deux lignes, ancienne et nouvelle, sont correctes.
– Supposons à présent que la valeur de NCOM soit modifiée **en dernier**, alors que MONTANT a déjà reçu sa nouvelle valeur. L'ancienne ligne de COMMANDE aura en parallèle été actualisée en fonction de cette nouvelle valeur. Lorsqu'on modifie NCOM, cette ligne ancienne est à nouveau actualisée par soustraction de la nouvelle valeur de MONTANT, ce qui est correct. Pas de problème pour la nouvelle ligne de COMMANDE.
– Il reste une troisième possibilité : la valeur de NCOM est modifiée **après** celle de QCOM et **avant** celle de NPRO (ou inversement), c'est-à-dire pour une valeur intermédiaire de MONTANT. Avant la modification de NPRO, l'ancienne ligne de COMMANDE aura été actualisée selon cette valeur intermédiaire, ce qui est prématuré. Lors de la modification de NPRO, l'ancienne ligne de COMMANDE est actualisée une seconde fois par soustraction de cette valeur intermédiaire, ce qui est correct. Pas de problème pour la nouvelle ligne de COMMANDE.

On constate donc que, quel que soit l'ordre d'exécution des déclencheurs, l'état final des données des tables DETAIL et COMMANDE est unique. La valeur de TOTAL_CLI, sera elle aussi recalculée trois fois, mais la valeur finale sera également unique .

Les déclencheurs élémentaires que nous avons rédigés sont donc bien **déterministes**, leur effet ne dépendant pas de leur ordre d'exécution.

Le code complet de cette application est disponible dans le fichier **Declencheurs - Gestion de la redondance.sql**.

A9.4.4 Application : gestion d'une nomenclature de produits

Cette application des structures de données cycliques, que nous avons introduite au chapitre 8 et étendue lors de l'étude des requêtes récursives dans la présente annexe, est une très belle illustration de l'usage intelligent des déclencheurs.

Nous nous intéresserons à un problème spécifique : le recalcul du prix des produits composés lorsque le prix à l'unité d'une matière première est modifié. Dans un tel cas, il faut mettre à jour le prix de tous les produits dont cette matière première est un composant direct ou indirect. Cette mise à jour est de toute évidence récursive et sera donc effectuée par un trigger récursif.

Modification d'une matière première

On définira deux déclencheurs selon que la nouvelle valeur de PRIX_U est *null* ou non *null* (script A9.24)

```
-- Assignation d'une valeur NULL
create trigger UPDT_PRIX_PRIMAIRE_NULL
instead of update of PRIX_U on MATIERE_PREMIERE
for each row
when new.PRIX_U is null
begin
  update PRODUIT
  set    PRIX_U = null
  where  NPRO = new.NPRO;
end;

-- Assignation d'une valeur non NULL
create trigger UPDT_PRIX_PRIMAIRE_NOTNULL
instead of update of PRIX_U on MATIERE_PREMIERE
for each row
when new.PRIX_U is not null
begin
  update PRODUIT
  set    PRIX_U = new.PRIX_U
  where  NPRO = new.NPRO;
end;
```

Figure A9.24 - Application de la modification du prix d'une matière première dans la table PRODUIT

Propagation des modifications dans la hiérarchie des produits

Il s'agit ensuite de propager la modification du prix d'une matière première vers tous les produits qui utilisent directement ou indirectement cette matière. Ici encore, on distinguera les cas des valeurs *null* et non *null* (A9.25).

– Si une valeur *null* a été assignée, alors tous les produits utilisateurs directs auront également un PRIX_U mis à *null*. Cette mise à *null* va redéclencher récursivement ce déclencheur sur les produits du niveau supérieur.

– Si une valeur non *null* a été assignée, alors on recalculera la valeur de PRIX_U de chacun de ses produits utilisateurs directs dont **tous les composants** ont une valeur non *null* de PRIX_U.

```
-- Assignation d'une valeur NULL
create trigger UPDT_COST2
after update of PRIX_U on PRODUIT
for each row
when new.PRIX_U is null
begin
  update PRODUIT
  set    PRIX_U = null
  where  NPRO in (select COMPOSE
                  from   COMPOSITION
                  where  COMPOSANT = old.NPRO)$;$
end;

-- Assignation d'une valeur non NULL
create trigger UPDT_COST1
after update of PRIX_U on PRODUIT
for each row
when new.PRIX_U is not null
begin
  update PRODUIT
  set    PRIX_U =  (select sum(B.PRIX_U*C.QTE)
                    from   COMPOSITION C, PRODUIT B
                    where  C.COMPOSE = PRODUIT.NPRO
                    and    C.COMPOSANT = B.NPRO)
  where NPRO in (select COMPOSE
                 from   COMPOSITION C
                 where  C.COMPOSANT = new.NPRO
                 and not exists(
                         select 1
                         from   PRODUIT P, COMPOSITION CC
                         where  CC.COMPOSE = C.COMPOSE
                         and    P.NPRO = CC.COMPOSANT
                         and    P.PRIX_U is null))$;$
end;
```

Figure A9.25 - Propagation de la modification du prix d'un produit

Application : calcul initial du prix des produits composants

Ce déclencheur conduit à une application intéressante : le (re)calcul des prix après insertion et suppression des données de matières premières, par exemple après création de la base de données et chargement des valeurs initiales de PRIX_U des matières premières (opérations `insert` et `delete`). L'exécution de la simple requête ci-dessous va entraîner le garnissage automatique de toutes les valeurs manquantes de PRIX_U :

```
update MATIERE_PREMIERE set PRIX_U = PRIX_U;
```

À comparer avec la requête de mise à jour A9.16.

A9.4.5 Journalisation des opérations de modification

On désire garder une trace précise des opérations de modification des données. Dans ce but, on définit une table JOURNAL dans laquelle on va consigner l'historique des opérations insert, update et delete affectant les tables de la base de données. Cet historique comporte les informations suivantes :

- **LogID** : numéro de séquence de l'entrée du journal (identifiant primaire)
- **DataTime** : date (`aaaa-mm-jj`) et heure (`hh:mm:ss.mmm`) de l'opération
- **Operation** : nature de l'opération (`insert`, `update`, `delete`)
- **Object** : nom de la table ayant subi la modification
- **Arguments** : caractéristiques de l'opération (par exemple, identification de la ligne, ancienne/nouvelle valeurs)

Le déclencheur du script A9.26 crée une ligne dans le journal après chaque insertion dans la table DETAIL. La fonction `current_timestamp_full()` est une UDF[5] (user-defined function) qui renvoie, sous un format standardisé, la valeur de l'instant présent avec une précision d'une milliseconde, suffisante pour les besoins de cette application.

Le fichier **Declencheurs - Gestion de la redondance + Log.sql** illustre l'utilisation de cet historique pour la gestion de la redondance.

```
create trigger TRG_TRACE_DET_INS
after insert on DETAIL
for each row
begin
  insert into JOURNAL(DateTime,Operation,Object,Arguments)
         values(current_timestamp_full(),
                'delete',
                'DETAIL',
                'NCOM = '||old.NCOM||' & NPRO = '||old.NPRO)$;$
end;
```

Figure A9.26 - Enregistrement de l'historique des insertions dans la table DETAIL

5. Cette fonction est un membre de la bibliothèque d'UDF **SQLiteUDFlib.py** de la distribution SQLfast. Cette bibliothèque extensible est à la disposition des utilisateurs, ... sous leur entière responsa-bilité !

A9.5 LE CATALOGUE

Cette section a pour objectifs d'appliquer et d'étendre le concept de catalogue au contexte de SQLfast.

A9.5.1 Le catalogue SQLfast

Dans l'environnement SQLfast, le dictionnaire de la base de données courante est créé par la commande **`createDictionary`**. Le dictionnaire comprend 6 tables et une vue. Outre les tables SYS_TABLE, SYS_COLUMN, SYS_KEY et SYS_KEY_COMP décrites au chapitre 9, le catalogue SQLfast comprend les tables SYS_INDEX et SYS_INDEX_COMP, décrivant les index de manière similaire aux clés. La table SYS_TABLE ne comprend cependant pas de colonne CREATOR. La vue SYS_INTER_TABLE décrit le graphe des tables à partir des couples *clé étrangère/identifiant*. Les colonnes étant légèrement différentes de celles décrites au chapitre 9, on en donne le rélevé et la définition ci-après.

table **SYS_TABLE**

- TableID : identifiant technique de la table
- TableName : nom de la table
- TableType : type d table (base = table de base, view = vue SQL)

table **SYS_COLUMN**

- ColID : identifiant technique de la colonne
- TableID : identifiant technique de la table de la colonne
- ColSeq : numéro de séquence de la colonne dans sa table
- ColName : nom de la colonne
- ColType : type de données de la colonne
- ColLen1 : longueur des valeurs
- ColLen2 : nombre de décimales pour les valeurs numériques
- ColNull : indique si la colonne est facultative (N = non, Y = oui)
- DefVal : valeur par defaut
- PKseq : si la colonne appartient à l'identifiant primaire, son numéro d'ordre

table **SYS_KEY**

- KeyID : identifiant technique de la clé
- KeyType : type de la clé (PK = primary key; UN = unique, FK = foreign key)
- TableID : identifiant technique de la table de la clé
- KeyName : nom de la clé
- TableTarget : identifiant de la table cible pour une FK
- KeyTarget : identifiant de la clé (PK ou UN) cible pour une FK
- onDelete : delete mode pour une FK
- onUpdate : update mode pour une FK

- Match : match mode pour une FK
- KeyNull : indique si les colonnes de la clés sont toutes facultatives (N = non, Y = oui)

table **SYS_KEY_COMP**

- KeyCompID : identifiant technique du composant de clé
- KeyID : identifiant technique de la clé du composant
- CompSeq : numéro de séquence du composant dans sa clé
- ColID : identifiant technique de la colonne du composant
- ColName : nom de la colonne du composant
- ColNull : indique si la colonne est facultative (N = non, Y = oui)
- CompTarget : identifiant technique du composant cible de l'identifiant cible pour une FK

view **SYS_INTER_TABLE**

- SourceTable : nom de la table de la clé étrangère
- SourceKey : identifiant technique de la clé étrangère
- TargetKey : identifiant technique de l'identifiant cible de la clé étrangère
- TargetTable : nom de la table cible de la clé étrangère

Contenu de la vue SYS_INTER_TABLE pour la base de données **CLICOM.db** :

```
+-------------+-----------+-----------+-------------+
| SourceTable | SourceKey | TargetKey | TargetTable |
+-------------+-----------+-----------+-------------+
| COMMANDE    | 5         | 1         | CLIENT      |
| DETAIL      | 6         | 2         | PRODUIT     |
| DETAIL      | 7         | 3         | COMMANDE    |
+-------------+-----------+-----------+-------------+
```

Ces tables sont temporaires et disparaissent au moment de la clôture de la base de données. Elle sont cependant mises à jour suite aux opérations DDL. Pour des raisons de facilité d'utilisation certaines données sont redondantes. Le petit script A9.27 affiche le contenu des principales tables et vues d'un catalogue.

```
openDB CLICOM.db;
  createDictionary;
  select * from SYS_TABLE
  select * from SYS_COLUMN
  select * from SYS_KEY
  select * from SYS_KEY_COMP
  select * from SYS_INTER_TABLE
closeDB;
```

Figure A9.27 - Affichage du contenu d'un catalogue

A9.5.2 Utilisation du catalogue

Les requêtes du chapitre 9 illustrant l'usage des données du catalogue se traduisent aisément :

```
openDB CLICOM.db;
createDictionary;
  select ColName,ColType,ColLen1,ColNull
  from   SYS_COLUMN C, SYS_TABLE T
  where  C.TableID = T.TableID
  and    TableName = 'DETAIL';

  select TableName
  from   SYS_TABLE
  where  TableID  in (select TableID
                      from   SYS_COLUMN
                      where  ColName like 'NCOM%')
                      and    TableType = 'base';
closeDB;
```

Figure A9.28 - Deux requêtes typiques sur le catalogue

A9.5.3 Exploitation du graphe des tables

La vue SYS_INTER_TABLE nous offre une bonne occasion de rédiger des requêtes récursives particulièrement utile dans la perspective de la gestion de la base de données. Cherchons par exemple à identifier toutes les tables dont le contenu est susceptible d'être touché, directement ou indirectement, par la suppression ou la modification d'une ligne d'une certaine table. Par exemple, la suppression d'une ligne de la table CLIENT va mettre en cause l'intégrité référentielle de la table COMMANDE, et, transitivement, celle de la table DETAIL. Le script de la figure A9.29 demande à l'utilisateur de sélectionner une table du schéma[6] puis affiche l'ensemble des tables dépendant de celle-ci.

```
openDB CLICOM.db;
  createDictionary;
  ask table = Table:[select TableName from SYS_TABLE];
  with recursive R(Rang,Nom)
  as (values (1,'$table$')
      union all
      select R.Rang+1, IT.SourceTable
      from SYS_INTER_TABLE IT, R
      where IT.TargetTable = R.Nom)
  select distinct * from R;
closeDB;
```

Figure A9.29 - Recherche des tables dépendant d'une table cible

6. Le dialogue est un peu simpliste. Il conviendrait de vérifier que l'utilisateur a bien sélectionné une table et a cliqué sur le bouton OK.

La saisie du nom CLIENT nous donnera le résultat suivant :

```
+------+----------+
| Rang | Nom      |
+------+----------+
| 1    | CLIENT   |
| 2    | COMMANDE |
| 3    | DETAIL   |
+------+----------+
```

A9.5.4 Rétro-ingénierie d'un schéma SQL

Le terme *rétro-ingénierie* est utilisé ici dans un sens très limité. Nous cherchons simplement à reconstituer le code SQL-DDL du schéma d'une table à partir du contenu du catalogue. Une présentation plus complète du processus de rétro-ingénierie fait l'objet du chapitre 22.

Le code sera produit par une unique requête **`select`**. La forme générale est la suivante :

```
select 'create table '||T.TableName||'('||
    <code des colonnes>||
    <code des identifiants primaires>||
    <code des identifiants secondaires>||
    <code des clés étrangères>||
   ');' as "create table"
from   SYS_TABLE T
where  T.TableType = 'base';
```

Nous construirons le code des colonnes au moyen de la fonction `group_concat` appliquée à l'ensemble des descriptions des colonnes de la table **`T`** :

<code des colonnes> ≡

```
(select group_concat(
              ColName||' '||
              <code de type>||<code not null>||<code default>,',')
 from   SYS_COLUMN C
 where  TableID = T.TableID)
```

A titre d'exemple, développons le bloc **<code de type>** :

```
case C.ColType
when 'char'    then 'char('||cast(ColLen1 as char)||')'
when 'varchar' then 'varchar('||cast(ColLen1 as char)||')'
when 'decimal' then 'decimal('||cast(ColLen1 as char)
                               ||','||cast(ColLen2 as char)
                               ||')'
else C.ColType
end
```

Les autres composants se construisent selon les mêmes principes. Le code complet est disponible dans le fichier **SQL-Generation.sql**

A9.6 L'INJECTION DE CODE SQL

<à rédiger à partir de :
https://staff.info.unamur.be/dbm/Documents/Tutorials/SQLfast/SQLfast-Tuto42-SQL-injection.pdf>

A9.7 INTERFACE PYTHON - BASES DE DONNÉES

Les distributions standard de Python incluent plusieurs modules d'accès à des bases de données, ou plus généralement à des données persitantes, tels que `dbm`, `anydbm`, `Gadfly` ou `sqlite3`). Le plus complet est `sqlite3`, dédié à SQLite. Il offre une interface conforme au standard de fait DB-API 2.0, fonctionnellement assez proche de JDBC mais dont la syntaxe est plus simple. Il existe des interfaces DB-API Python pour la plupart des SGBD SQL, voire certains SGBD NoSQL (Cassandra/ CQL, Berkeley DB).

A9.7.1 Bases de données SQLite

SQLite[7] est un SGBD relationnel développé par D. Richard Hipp. Des interfaces ont été développées pour la plupart des langages de programmation.

Toutes les données d'une base de données SQLite 3 (tables, index, catalogue, etc.) sont stockées dans un unique fichier. Nous conviendrons de lui donner l'extension **`.db`**. Le SGBD SQLite ne s'exécute pas comme un serveur autonome selon une architecture *client-serveur* (chapitre 5) mais comme une simple bibliothèque de fonctions intégrée à chaque programme d'application. Cette caractéristique définit les principaux avantages mais aussi les limitations des bases de données SQLite :

– L'utilisation de SQLite ne requiert aucune installation, ni native (utilisation autonome en dehors de Python), ni pour son utilisation via Python.

– L'unité d'accès en mise à jour est le fichier (seule exception : les bases de données *in-memory*, non persistantes). Un seul programme à la fois est autorisé à effectuer des modifications sur une base de données. En revanche, la lecture partagée est autorisée. SQLite offre cependant un système de transactions relativement riche.

– SQLite ne possède pas de fonctions de gestion des utilisateurs ni de contrôle d'accès. D'une manière générale, il ne fournit pas de fonctions d'administration.

– Le code SQLite est extrêmement compact : moins de 700 Ko pour la version de 2015. Ceci explique l'intégration de SQLite dans la plupart des applications ayant à gérer des données d'une certaine complexité, et en particulier les applications mobiles. Il fait d'ailleurs partie intégrante des distributions Linux, Android, iOS et Mac OS.

Les fonctions d'accès aux bases de données SQLite sont regroupées dans le module **`sqlite3`** des distributions standard de Python. En outre, ces distributions incluent le

7. https://www.sqlite.org/

SGBD SQLite lui-même, de sorte qu'aucune installation supplémentaire n'est nécessaire.

Nous examinerons à présent la manière dont les principales opérations sur une base de données se traduisent en Python/DB-API dans sa version SQLite. Nous utiliserons la syntaxe Python 2.7.

Références de base

– **SQLite** : https://www.sqlite.org
– **DB-API 2.0** : *PEP 0249 -- Python Database API Specification v2.0*, https://www.python.org/dev/peps/pep-0249
– **sqlite3** : *sqlite3 — DB-API 2.0 interface for SQLite databases*, https://docs.python.org/2/library/sqlite3.html

A9.7.2 Création d'une base de données

Une première remarque : tout programme et fonction opérant sur une base de données SQLite doit inclure l'instruction suivante, qui rend accessibles les fonctions de SQLite 3 :

```
import sqlite3
```

La création d'une base de données s'effectue par la fonction **`connect`** du module sqlite3, dont l'argument est le nom d'une base de données. Celle-ci crée un objet gérant cette connexion :

```
db = sqlite3.connect(nomBD)
```

Cet objet offre quelques méthodes globales telles que la création et la fermeture d'un curseur, commit, rollback et bien entendu la fermeture de la connexion.

Attention, la fonction **`connect`** se contente d'ouvrir une base de données de nom nomBD si celle-ci existe déjà. Il est donc prudent de tester son existence, par exemple par la petite fonction **`existeFichier`** (A9.30). On écrira donc :

```
import sqlite3
nomBD = 'CLICOM.db'
if existeFichier(nomBD):
    print u'La base de données '+nomBD+u' existe déjà'
else:
    db = sqlite3.connect(nomBD)
```

```
def existeFichier(nomDB):
    import os.path
    return os.path.isfile(nomDB)
```

Figure A9.30 - Vérifier l'existence d'un fichier

A9.7.3 Ouverture d'une base de données

La connexion à une base de données s'effectue également par la fonction **`connect`** :

```
import sqlite3
nomBD = u'CLICOM.db'
db = sqlite3.connect(nomBD)
```

Pour éviter la création de cette base de données si elle n'existe pas :

```
if not existeFichier(nomBD):
    print 'La base de données '+nomBD+' est inconnue'
else:
    db = sqlite3.connect(nomBD)
```

A9.7.4 Extraction de données

Conformément aux principes des interfaces dynamiques, la requête est une chaîne de caractères spécifiée par une constante ou assignée à une variable :

```
requete = u'select NCLI,NOM,LOCALITE from CLIENT'
```

Le canevas de base

L'exécution de cette requête nécessite l'existence d'un curseur :

```
c = bd.cursor()
```

dont la méthode **`execute`** permet d'exécuter la requête :

```
c.execute(requete)
```

Dès cet instant, le curseur représente le résultat de cette exécution. Il existe pour y accéder de multiples manières. La plus simple est de traiter le curseur comme un itérateur qui livre une suite de tuples Python :

```
for ligne in c:
    print u'%-6s  %-10s  %-10s' % ligne
```

ce qui nous donne :

```
B062    GOFFIN      Namur
B112    HANSENNE    Poitiers
C400    FERARD      Poitiers
etc.
```

Pour résumer :

```
requete = u'select NCLI,NOM,LOCALITE from CLIENT'
c = bd.cursor()
c.execute(requete)
for ligne in c:
    print u'%-6s  %-10s  %-10s' % ligne
```

Variante plus concise encore :

```
requete = u'select NCLI,NOM,LOCALITE from CLIENT'
c = bd.cursor()
for ligne in c.execute(requete):
    print u'%-6s  %-10s  %-10s' % ligne
```

Tant qu'il n'est pas supprimé (**`c.close()`**), un curseur peut servir à l'exécution de plusieurs requêtes successives.

Les méthodes fetchall() et fetchone() des curseurs

La méthode **fetchall()** extrait le résultat sous la forme d'une *liste de tuples* :

```
resultat = c.fetchall()
for ligne in resultat:
    print u'%-6s  %-10s  %-10s' % ligne
```

La méthode **`fetchone()`** extrait le *tuple suivant* du résultat :

```
ligne = c.fetchone()
while ligne is not None:
    print u'%-6s  %-10s  %-10s' % ligne
    ligne = c.fetchone()
```

ou encore :

```
while True:
    ligne = c.fetchone()
    if ligne is None:
        break
    print u'%-6s  %-10s  %-10s' % ligne
```

Les curseur implicites

Les méthodes **`execute()`**, **`fetchone()`** et **`fetchall()`** existent aussi pour les objets de connexion. Les requêtes précédentes peuvent alors s'écrire sans curseur :

```
requete = u'select NCLI,NOM,LOCALITE from CLIENT'
bd.execute(requete)
for ligne in bd:
    print u'%-6s  %-10s  %-10s' % ligne

for ligne in bd.execute(requete):
    print u'%-6s  %-10s  %-10s' % ligne

resultat = bd.fetchall()

ligne = bd.fetchone()
```

En réalité, un curseur implicite temporaire est créé pour l'exécution de la requête. Il s'agit donc d'un raccourci d'écriture.

Requêtes paramétrées

Lorsqu'une requête comporte un paramètre, celui-ci peut apparaître sous la forme d'une constante *explicite* :

```
requete = u"select * from CLIENT where NCLI = 'C400'"
```

ou d'une constante *injectée* :

```
N = u'C400'
requete = u"select * from CLIENT where NCLI = '" + N + u"'"
```

Mais on sait que cette technique peut s'avérer dangereuse car elle permet des attaques par *injection de code SQL* (section 9.11.6). Il est préférable d'utiliser la technique consistant à indiquer dans la requête l'emplacement des paramètres par des symboles "**?**" puis de fournir leurs valeurs séparément sous la forme d'un tuple. L'exemple JDBC complet de l'extraction des données des clients de Poitiers de la section 9.11.4 s'écrirait comme suit, en Python/SQLite 3 :

```
db = sqlite3.connect(u'CLICOM.db')
requete = u'select NCLI,NOM from CLIENT where LOCALITE = ?'
varLOC = u'Poitiers'
c = bd.cursor()
c.execute(requete,(varLOC,))
for row in c:
    print u'%-6s  %-10s' % row
```

Il existe une variante utilisant des noms de variables au lien de marqueurs "?", mais elle s'avère assez lourde à l'usage. Les variables sont définies dans la requête SQL et l'assignation de valeurs est spécifiée dans un dictionnaire :

```
valeurs = {u'ncli':u'C400'}
c.execute(u'select * from CLIENT where NCLI = :ncli',valeurs)
for ligne in c:
    print u'%-6s  %-10s  %-10s' % ligne
```

Noms des colonnes d'un résultat

L'attribut **`description`** d'un curseur renvoie la liste des noms des colonnes du résultat de l'exécution d'une requête **`select`**[8] (en fait une liste de septuplets dont le premier élément est ce nom) :

```
c.execute(u'select * from CLIENT')
for nom in c.description:
```

8. Y compris lorsque ce résultat est vide. Une exception (= bug) : lorsque la composante d'initialisation d'une requête **`with`** renvoie un résulat vide, l'attribut description a la valeur **None**.

```
print nom[0]
```

Remarque.

Le standard DB-API ne propose pas d'instruction de précompilation de requête (**`prepare`**). Cette opération est implicite lors de l'exécution de la requête. Cependant, SQLite mémorise en cache les dernières instructions exécutées pour éviter de les réanalyser.

A9.7.5 Création des structures de données

Les opérations **`create`**, **`drop`** and **`alter`**[9] sont exécutées de la même manière que les opérations d'extraction de données:

```
create = u'create table CLIENT('+
         u'NCLI char(10) not null,'+
         u'NOM  char(32) not null,'+
         u'LOCALITE char(30) not null,'+
         u'primary key (NCLI))'
c = bd.cursor()
c.execute(create)
```

A9.7.6 Modification des données

Il en est de même des opérations **`insert`**, **`update`** et **`delete`**, que nous illustrerons par diverses variantes syntaxiques :

```
insert = u"insert into CLIENT values('B062','GOFFIN','Namur')"
c.execute(insert)

insert = u"insert into CLIENT values(?,?,?)"
valeurs = (u'B062',u'GOFFIN',u'Namur')
c.execute(insert,valeurs)

update = u"update CLIENT set NOM = ? where NCLI = ?)"
valeurs = (u'ADELIN',u'B062')
c.execute(update,valeurs)

c.execute(u'delete from CLIENT')
```

Lorsque les valeurs sont constituées d'une *liste*[10] *de tuples* au lieu d'un simple tuple, la méthode **`executemany()`** exécute l'opération SQL pour chacun de ces tuples :

```
Lval = [(u'B512',u'GILLET',u'Toulouse'),
        (u'C003',u'AVRON',u'Toulouse')]
c.executemany(u'insert into CLIENT values(?,?,?)',Lval)
```

9. SQLite 3 n'offre pour l'instant qu'une variante réduite de l'instruction **`alter`** : ajout d'une colonne et renommage d'une table.
10. Plus généralement un itérateur.

A9.7.7 Gestion des exceptions et transactions

Toute erreur survenant lors de l'exécution d'une requête SQL se traduit par une exception provoquant l'arrêt du programme. Il est donc recommandé d'inclure les les instructions du module sqlite3 dans une structure de gestion des exceptions :

```
delete = u'delete from KLIENT where NCLI = ?'
ncli = u'C400'
try:
    c.execute(delete,(ncli,))
except sqlite3.Error as err:
    print u"Erreur [" + unicode(err.args[0]) + u"]
```

Le nom de la table étant erroné, on obtiendra :

```
Erreur [no such table: KLIENT]
```

SQLite dispose des méthodes suivantes, contrôlant la terminaison de transactions :

```
bd.commit()
bd.rollback()
```

Ainsi que de l'instruction d'ouverture d'une transaction (on notera l'asymétrie malheureuse) :

```
c.execute(u'begin transaction')
```

Le fragment ci-dessous expérimente le comportement de SQLite lors de la survenance d'un incident inopiné, ici, une division par `0`. En remplaçant l'instruction `X = 0` par `X = 1`, on évite l'incident. La dernière instruction (`rollback`) permet de restaurer l'état initial des données à la fin de l'expérience.

```
def afficherCLIENT(con,mess=u'\n'):
    print u'\n'+mess
    for ligne in con.execute(u'select * from CLIENT'):
        print u'%-6s  %-10s  %-10s' % ligne

X = 0

afficherCLIENT(bd,u'CLIENT avant suppression de C400')

try:
    bd.execute(u"delete from CLIENT where NCLI='C400'")
    afficherCLIENT(bd,u'CLIENT après suppression de C400')
    X = 100/0  # On provoque artificiellement une erreur
    bd.commit()
    afficherCLIENT(bd,u'CLIENT après commit')
except Exception as err:
    print u'\nErreur détectée : '+unicode(err[0])
    afficherCLIENT(bd,u'CLIENT après une erreur provoquée')
    bd.rollback()
    afficherCLIENT(bd,u'CLIENT après rollback')

afficherCLIENT(bd,u'CLIENT état final')
bd.rollback()
```

Le mécanisme des transactions SQLite 3 offre de nombreuses possibilités qu'il serait trop long de décrire ici. Une description approfondie de ces transactions est disponible (en anglais) à l'adresse suivante[11] :

http://www.info.fundp.ac.be/~dbm/Documents/Tutorials/SQLfast/SQLfast-TutoA8-SQLite-Transactions.pdf (*draft version*)

A9.7.8 Les métadonnées

Traditionnellement, la description du schéma d'une base de données est stockées dans les tables du catalogue. Les choses sont un peu différentes en SQLite.

La description des objets majeurs du schéma (tables, index, vues, tiggers) se trouve dans la table **sqlite_master**, dont les colonnes fournissent, respectivement,

- le type de l'objet (**type**)
- le nom de l'objet (**name**)
- le nom de la table dont dépend l'objet (**tbl_name**)
- le numéro de la première page de la table (**rootpage**)
- le code SQL DDL qui a servi à créer l'objet (**sql**)

La recherche des tables et des vues de la base de données s'obtient de la manière suivante :

```
c.execute(u'''select type,name
              from   sqlite_master
              where  type in ('table','view')''')
for table in c:
    print u'%-5s  %-16s' % table
```

Les informations décrivant les colonnes sont livrées par une pseudo-table matérialisée par la fonction **PRAGMA** `table_info` :

```
c.execute(u"PRAGMA table_info('CLIENT')")
for colonne in c:
    print u'%-16s' % colonne[1]
```

Cette fonction produit des tuples constitués comme suit :

- numéro d'ordre de la colonne dans la table (**cid**)
- nom de la colonne (**name**)
- type de valeurs (**type**)
- contrainte *not null* (**notnull**)
- valeur par défaut (**dflt_value**)
- numéro d'ordre dans l'identifiant primaire, ou 0 (**pk**)

11. Ce document est encore en cours de rédaction mais est jugé suffisamment complet dans son état actuel.

En emboîtant ces deux boucles, on imprime une description des tables et des colonnes :

```
c1 = bd.cursor()
c2 = bd.cursor()
c1.execute(u"select name
           from   sqlite_master
           where  type = 'table'")
tables = c1.fetchall()
for table in tables:
    print u'table %s' % table[0]
    c2.execute(u"PRAGMA table_info('"+table[0]+u"')")
    for colonne in c2:
        print u'   %s' % colonne[1]
```

dont le résultat pourrait être :

```
table CLIENT
   NCLI
   NOM
   LOCALITE
. . .
```

On observe que l'extraction des données des tables s'exprime par un **`fetchall`**. En effet, l'emboîtement direct des deux boucles produit des effets indésirables.

Il existe trois autres fonctions PRAGMA :

- **`index_list`**`('CLIENT')`, donnant les index d'une table (dont on dérive tous les identifiants).
- **`index_info`**`('sqlite_autoindex_CLIENT_1')`, donnant les composants d'un index.
- **`foreign_key_list`**`('CLIENT')`, donnant les composants des clés étrangères. On en dérive avant tout les clés étrangères elles-mêmes.

L'instruction **`createDictionary`** de la version de base de SQLfast exploite ces informations pour construire les tables SYS_TABLE, SYS_COLUMN, SYS_KEY, SYS_KEY_COMP, SYS_INDEX et SYS_INDEX_COMP.

A9.8 L'INFORMATION INCOMPLÈTE

<à rédiger>

A9.9 EXERCICES DU CHAPITRE 9

A9.1 Développer, à l'aide des concepts étudiés dans cet ouvrage, un mécanisme qui permet de limiter l'accès à une table par un utilisateur déterminé à des moments bien déterminés, par exemple du lundi au vendredi, de 9h à 17h.

Solution

On suggère de définir une vue limitant l'accès aux plages horaires désirées.

```
create view T_CLIENT(NCLI, NOM, ...) as
select NCLI, NOM, ...
from CLIENT
where extract(weekday from current_date) between 1 and 5
and   extract(hour from current_time) between 9 and 17;
revoke select on CLIENT from U_MERCIER:
grand select on T_CLIENT to U_MERCIER;
```

A9.2 Exprimer l'opérateur `except` à l'aide d'une jointure externe.

Solution

Exemple : liste des numéros des clients qui n'ont pas passé de commande :

```
select C.NCLI
from CLIENT C left outer join COMMANDE M
     on (C.NCLI = M.NCLI)
where NCOM is null;
```

A9.3 Ecrire une requête qui produise les supérieurs hiérarchiques, directs ou indirects, d'une personne dont on spécifie le numéro.

A9.4 Ecrire une requête qui donne, pour chaque table, le nombre de colonnes et la longueur maximum des lignes.

A9.5 Le catalogue que nous avons décrit dans le chapitre 9 (figures 9.4 et 9.5) diffère quelque peu des catalogues proposés par les SGBD. En particulier, les catalogues réels ont souvent une structure complexe destinée à améliorer les performances. Ecrire les requêtes qui garnissent les tables de la figure 9.5 à partir du contenu des tables du catalogue de votre SGBD. Pour fixer les idées, on pourra s'exercer sur la structure ci-dessous, limitée à la représentation des *clés* (identifiants et clés étrangères) :

SYS_KEY_COL					
TNAME	KTYPE	KEYID	CNAME	TARGTAB	TARGCOL
CLIENT	P	K1	NCLI		
COMMANDE	P	K2	NCOM		
COMMANDE	F	K3	NCLI	CLIENT	NCLI
PRODUIT	P	K7	NPRO		
DETAIL	P	K4	NCOM		
DETAIL	P	K4	NPRO		
DETAIL	F	K5	NCOM	COMMANDE	NCOM
DETAIL	F	K6	NPRO	PRODUIT	NPRO

Une ligne de cette table représente un composant d'une *clé*. Elle indique successivement le nom de la table de la *clé*, le type de la clé (*P*rimary, *F*oreign), son identifiant interne et le nom de la colonne; s'il s'agit d'une clé étrangère, elle indique en outre le nom de la table et de la colonne cibles.

A9.6 Le lecteur attentif aura observé que la table SYS_KEY_COL de l'exercice A9.5 n'est pas normalisée. On l'invite à repérer les dépendances fonctionnelles anormales puis à décomposer la table de manière à obtenir des fragments normalisés.

A9.7 Le dictionnaire (ou catalogue) de SQLfast comporte quelques redondances. Lesquelles ? Ecrire les requêtes qui permettent de restaurer les données redondantes dans l'hypothèse où elles auraient été corrompues suite à une fausse manoeuvre.

A9.8 Chaque table du dictionnaire de SQLfast comporte une colonne additionnelle **Ext** (pour *extension*), dont la signification est à la discrétion de ses utilisateurs. On propose de l'utiliser pour documenter chaque table et chaque colonne d'une base de données en lui associant une définition en langue naturelle. Cependant, ces informations seront perdues à la fermeture de la base de données. Développer une procédure simple qui permette de conserver cette documentation.

Suggestion. avant la fermeture, sauver ces informations sous la forme d'un script SQL constitué de requêtes **`update`** et à exécuter après l'ouverture utltérieure de cette base de données. Prévoir le fait que ces descriptions pourraient être attachées à des objets qui auraient disparu ou auraient changé de nom.

A9.9 Un dictionnaire SQLfast décrit, de manière temporaire, une seule base de données. Il serait intéressant de contruire un dictionnaire multibases qui décrirait de manière permanente plusieurs bases de données.

Suggestion. Ce dictionnaire multibases est une base de données au sens propre. SQLfast ne pouvant ouvrir qu'une seule base de données à la fois, le contenu du dictionnaire local est d'abord sauvé sous la forme d'un script de

chargement (**`insert`**). La base de données est ensuite fermée, le dictionnaire multibases est ouvert, l'ancienne version du schéma de la base locale y est supprimée et le script de chargement est exécuté.

A9.10 Lorsque le dictionnaire multibases est disponible, écrire un script qui identifie les différences entre deux schémas.

Suggestion. On ne peut évidemment comparer deux schémas s'ils sont *raisonnablement* similaires, tels que ceux de deux versions successives de la même base de données en évolution. Si les objets n'ont pas changé de nom d'une version à l'autre, la comparaison peut se faire par simple calcul de la différence (opérateur **`except`**) entre deux ensembles de noms. Si les noms peuvent avoir été modifiés, la comparaison est plus complexe. Il faut dans ce cas évaluer la similarité structurelle entre les objets : par exemple, deux tables sont *similaires* si elles possèdent *à peu près* les mêmes colonnes, les mêmes identifiants et les mêmes clés étrangères.

A9.11 Ecrire une requête qui met à jour la valeur de COMPTE de chaque client en en retirant le total des montants des commandes de la date courante.

A9.12 Proposer un schéma de tables équivalent à celui de la figure 2.7 établi selon les concepts relationnels objet. On remplacera les clés étrangères par des colonnes de référence et on représentera la table DETAIL par une colonne complexe (tableau de 20 lignes) de la table COMMANDE.

A9.13 La requête ci-dessous est tirée d'une application Access fournie par Microsoft dans la distribution d'Office. Les noms de tables et de colonnes ont été anonymisés mais la structure a été conservée. Réécrivez cette requête en exprimant les jointures sous une forme standard.

```
select distinct *
from S inner join (P inner join ((E inner join
(C inner join A on C.I = A.CI) on E.I = A.EI)
inner join B on A.I = B.OI) on P.I = B.PI) on S.I = A.S;
```

A9.14 Quel résultat ce petit script va-t-il produire ?

```
create table A(A1 integer,A2 integer);
create table B(A1 integer,B2 integer);
insert into A values(1,11),(2,22);
select * from A left outer join B;
```

Solution

```
+----+----+----+----+
| A1 | A2 | A1 | B2 |
+----+----+----+----+
| 1  | 11 | -- | -- |
| 2  | 22 | -- | -- |
+----+----+----+----+
```

A9.15 Au moment de leur entrée en fonction, les utilisateurs Jean et Andre ne jouissent d'aucun privilège sur la base de données de leur entreprise. Les utilisateurs U1, U2 et Jean exécutent ensuite les requêtes (valides) de gestion de privilèges suivantes, dans cet ordre :

```
U1:   grant select, update(QSTOCK) on PRODUIT to Jean
      with grand option;
U2:   grant select, update(QSTOCK,PRIX) on PRODUIT to Jean
      with grant option;
Jean: grant select, update(PRIX) to André;
U1:   revoke grant option for update(QSTOCK) on PRODUIT
      from Jean;
U2:   revoke update(QSTOCK) on PRODUIT from Jean;
```

Indiquer les privilèges des utilisateurs Jean et Andre après l'exécution de chaque requête de cette séquence. Poser les hypothèses additionnelles nécessaires.