Date de dernière modification *: 27/6/2015*

Annexe **22**

Rétro-ingénierie d'une base de données

Cette annexe approfondit et illustre des problèmes et des techniques mentionnés dans le chapitre 22.

A22.1 RECHERCHE DES IDENTIFIANTS D'UNE TABLE PAR ANALYSE DES DONNÉES

Le problème posé est le suivant. Soit une source de données externes, de qualité indéterminée, se présentant sous la forme d'une table. Pour intégrer ces données dans une base de données, il est nécessaire d'en connaître un minimum sur sa structure et en particulier de repérer ses identifiants. Nous avons vu au chapitre 22 (section 22.4) différentes techniques de détection de ces contraintes lorsqu'elles sont inconnues.

L'une d'elles consiste à *analyser les données* pour en déterminer les propriétés. Si nous observons, suite à cette analyse, que les valeurs d'une colonne sont uniques parmi les lignes de la table, alors nous pouvons conclure que cette colonne est un *identifiant potentiel* de la table. Bien sûr, nous n'avons pas de garantie absolue que cette propriété d'unicité est intentionnelle et non un accident statistique, en particulier si le volume analysé est faible. Néanmoins, nous pouvons en tout état de cause conclure que *si ces valeurs ne sont pas uniques*, alors *cette colonne ne constitue pas un identifiant*.[1]

1. Cette conclusion elle-même peut être sujet à caution. Si les données sont de piètre qualité, il n'est pas exclu qu'un identifiant soit réellement imposé mais que certaines données violent cette contrainte. Pour tenir compte de cette éventualité, on affectera les observations découlant de l'analyse des données d'un *degré de certitude*. Par exemple, s'il apparaît que 100 lignes parmi 100 000 violent une propriété d'unicité, on conclura soit que cette propriété n'est pas observée, soit qu'est observée à 99,9%.

La procédure de base

Nous allons dans cette section développer un petit script SQLfast fournissant toutes les combinaisons de colonnes dont les valeurs sont uniques, et qu'on peut considérer comme des identifiants potentiels.

Exprimons d'abord sous la forme d'un fragment de code le test d'unicité des valeurs d'une colonne, ou plus généralement d'un groupe G de colonnes d'une table T. La variable NLinT contient le nombre de lignes de la table T et la variable NvalG le nombre de valeurs distinctes des colonnes du groupe G :

```
extract NLinT = select count(*) from T;
extract NvalG = select count(*)
                from  (select distinct G from T);[2]
if ($NvalG$ = $NLinT$)
    write G est un identifiant potentiel de T;
```

Nous allons donc effectuer ce test pour tous les groupes de colonnes de la table T. Pour ce faire, nous générons tous les sous-ensembles (que nous appellerons *combinaisons*, pour simplifier) de colonnes de T, et, pour chacun d'eux, nous calculons le nombre de valeurs distinctes présentes dans la table. Si T possède NColT colonnes, alors de nombre de combinaisons est égal à $2^{NColT} - 1$. Par exemple, on dérive de la table CLIENT $2^6 - 1 = 63$ combinaisons à tester.

Le test complet d'une table inconnue comportant un grand nombre de lignes et de colonnes est évidemment irréaliste, ce qui nous oblige à affiner la procédure. D'abord, nous devons nous assurer que la table T dispose d'au moins un identifiant :

```
extract NLinTU = select count(*)
                 from  (select distinct * from T);
if ($NLinTU$ = $NLinT$)
    write T possède au moins un identifiant potentiel;
```

à défaut de quoi il est inutile de poursuivre l'analyse ! Si les lignes sont uniques, on va envisager quatre optimisations permettant de réduire le coût de l'analyse de la table.

Limitation de la taille des combinaisons

La plupart des identifiants véritables comportent un très petit nombre de colonnes : 1, 2, parfois 3 mais rarement plus. On propose donc de limiter la taille des combinaisons à une valeur inférieure à **NColT**. Nous pouvons dès lors écrire la procédure de génération des combinaisons à tester. Les combinaisons sont rangées dans la table temporaire **COMB** comportant deux colonnes : List, qui contient une combinaison sous forme d'une liste de noms de colonnes et Size qui indique le nombre de colonnes de la combinaison. Une propriété qui aura son importance : dans chaque liste, les noms sont ordonnés par ordre de déclaration dans leur table. Si deux noms appartiennent à deux listes, elle y apparaissent dans le même ordre.

2. La forme multiple `count(distinct A,B)` n'est malheureusement pas autorisée en SQL2.

Pour générer les combinaisons, nous avons besoin du dictionnaire de la base de données (`createDictionary`), dont la table SYS_COLUMN nous donne la liste des colonnes de chaque table.

Les combinaisons sont créées par une requête récursive (script A22.1) dont la CTE LISTES comporte trois colonnes : List, le contenu d'une combinaison, Last, le numéro d'ordre (ColSeq) dans la table SYS_COLUMN de la dernière colonne ajoutée à la liste courante et Size, le nombre de colonnes de la liste courante. Une liste est prolongée par une colonne dont le numéro d'ordre est supérieur à celui de la dernière colonne ajoutée (`S.ColSeq > L.Last`). On garantit ainsi que les noms sont uniques dans la liste et que celle-ci est ordonnée par numéros d'ordre croissants. Le processus d'extension des combinaisons se poursuit tant que les combinaisons existantes n'ont pas atteint la taille limite spécifiée par la variable `NColI` (`L.Size < $NColI$`).

```
openDB CLICOM.db;
createDictionary;
create temporary table COMB(List varchar(256),
                            Size integer);
extract idT = select TableID from SYS_TABLE
              where  TableName = 'CLIENT';
set NColI = 3;
with LISTS(List,Last,Size) as
  (select ColName,ColSeq,1 from SYS_COLUMN
   where  TableID = $idT$
     union all
   select L.List||','||S.ColName,S.ColSeq,L.Size + 1
   from   LISTS L,SYS_COLUMN S
   where  TableID = $idT$
   and    S.ColSeq > L.Last
   and    L.Size < $NColI$)
insert into COMB(List,Size)
   select List,Size
   from   LISTS;
```

Figure A22.1 - Génération des combinaisons de colonnes de CLIENT de taille limitée à NColI

Élagage des combinaisons non minimales

Le chapitre 3 a introduit le concept d'**identifiant minimal**. Si G est un identifiant de T, alors toute combinaison G' telle que $G \subset G'$ est aussi un identifiant, mais celui-ci est évidemment non minimal. Les seuls identifiants qui nous intéressent sont les identifiants minimaux. Ainsi donc, dès qu'un identifiant potentiel G a été trouvé, nous pouvons supprimer des combinaisons restant à tester toutes celles qui contiennent G. Si la colonne NCLI de la table CLIENT a été reconnue comme identifiant potentiel, alors les 31 combinaisons qui contiennent NCLI peuvent être supprimées sans examen.

Quelle est la manière la plus simple de repérer ces combinaisons à supprimer ? Pour déterminer si la combinaison (B,D) est compriss dans un groupe M plus grand ((A,B,C,D,E) par exemple), il suffit de vérifier que ce dernier comprend B, puis, plus loin à droite, D. Le prédicat SQL **`like`** est parfaitement adapté pour exprimer cette propriété. La chaîne `'A,B,C,D,E'` contient `'B'` suivi de `'D'` si :

```
'A,B,C,D,E' like '%B%D%'
```

Le masque `'%B%D%'` se calcule aisément de la combinaison `'B,D'` présente dans COMB par l'expression SQL, exprimé pour la combinaison **G** :

```
'%'||replace(G,',','%')||'%'
```

En fait, ces masques sont précalculés et insérés dans la table COMB par la requête récursive décrite ci-avant. COMP comporte donc en réalité trois colonnes : List, Pattern (le masque) et Size. Le contenu de COMB pour la table CLIENT et la limite `NColI = 3` se présente comme suit :

List	Pattern	Size
CAT	%CAT%	1
NOM	%NOM%	1
NCLI	%NCLI%	1
COMPTE	%COMPTE%	1
ADRESSE	%ADRESSE%	1
NOM,CAT	%NOM%CAT%	2
LOCALITE	%LOCALITE%	1
NCLI,CAT	%NCLI%CAT%	2
...	...	...
NOM,LOCALITE,COMPTE	%NOM%LOCALITE%COMPTE%	3
ADRESSE,LOCALITE,CAT	%ADRESSE%LOCALITE%CAT%	3
NCLI,LOCALITE,COMPTE	%NCLI%LOCALITE%COMPTE%	3
NOM,ADRESSE,LOCALITE	%NOM%ADRESSE%LOCALITE%	3
NCLI,ADRESSE,LOCALITE	%NCLI%ADRESSE%LOCALITE%	3
ADRESSE,LOCALITE,COMPTE	%ADRESSE%LOCALITE%COMPTE%	3

Pour exploiter cette optimisation, nous devons tester les combinaisons par ordre croissant de taille. Dans le script A22.2, la boucle principale est pilotée par la taille des combinaisons, len, de `1` à `NColI` (une sortie prématurée est prévue dès que la table COMB est vide).

Chaque combinaison de taille `len` est évaluée et, si elle passe avec succès le test d'identifiant potentiel, est stockée dans une table temporaire ID, de même structure que COMB. Lorsque toutes les combinaisons de taille `len` ont été évaluées, elles sont supprimées de COMB. En outre, toutes les combinaisons de COMB qui contiennent un des identifiant potentiels de taille `len` sont également supprimées.

```
create temporary table ID(  List    varchar(256),
                            Pattern varchar(256),
                            Size    integer);
for len = [1,$NColI$];
  for lst,pat,siz = [select List,Pattern,Size from COMB
                      where  Size = $len$ order by List];
    extract Nval = select count(*)
                   from (select distinct $lst$ from $nomT$);
    if ($Nval$ = $NLinT$)
        insert into ID values('$lst$','$pat$',$siz$);
  endfor;
  delete from COMB where Size = $len$;
  for pat = [select Pattern from ID where Size = $len$]
    delete from COMB where  List like '$pat$';
  extract Nc = select count(*) from COMB;
  if ($Nc$ = 0) exit;
endfor;
```

Figure A22.2 - Repérage des identifiants potentiels

Appliquée à la table CLIENT, cette procédure produit le résultat ci-dessous :

```
+------------------------+
| Identifiants de CLIENT |
+------------------------+
| ADRESSE                |
| NCLI                   |
| NOM,CAT                |
| NOM,COMPTE             |
| NOM,LOCALITE           |
| LOCALITE,CAT,COMPTE    |
+------------------------+
```

Ce résultat peut surprendre par la présence d'identifiants non pertinents mais il s'explique aisément par le nombre très réduit de lignes de la table CLIENT. Il est évident que dans une table réelle comportant plusieurs dizaines ou centaines de milliers de lignes, la plupart des identifiants potentiels non significatifs vont disparaître.

Exclusion a priori des colonnes non pertinentes

L'examen du résultat que nous venons d'obtenir montre que certaines colonnes n'auraient jamais dû être prises en compte pour générer les combinaisons à évaluer. Tel est le cas de la colonne COMPTE, dont on peut admettre qu'elle ne peut faire partie d'aucun identifiant de CLIENT. Il en sera de même des colonnes QCOM, PRIX et QSTOCK des autres tables, dont on voit mal le rôle qu'elles pourraient jouer dans des identifiants significatifs. Les exclure dès le départ va réduire le nombre de combinaisons à évaluer.

Cette optimisation repose sur une compréhension minimale de la signification des colonnes. Elle est évidemment inapplicable dans le cas de données totalement inconnues. Tout au plus peut-on admettre que des colonnes de type `float`, `real`, `decimal` avec des décimales, `BLOB` ou `CLOB` feront rarement partie d'un identifiant.

La mise en oeuvre de cette optimisation est aisée. Pour la table à analyser, une boite de dialogue comportant une case à cocher par colonne est construite à partir de la table SYS_COLUMN. Après sélection par l'utilisateur des colonnes à prendre en compte, les colonnes à ignorer sont supprimées de la table SYS_COLUMN.[3] La suite de la procédure est identique à la précédente.

Analyse d'un échantillon réduit de la table

Si la table à analyser est très volumineuse, ne serait-il pas utile de travailler d'abord sur un sous-ensemble réduit de ses lignes ? Et dans ce cas, quelles informations sur la table complète peut-on tirer de cette analyse partielle ?

L'analyse d'une table, complète ou partielle, conduit à partitionner une collection de combinaisons de colonnes en deux sous-ensembles : les *identifiants potentiels* et les *combinaisons non uniques*. Supposons que nous ayons effectué ce classement sur une partie réduite de la table selon la procédure (optimisée) développée ci-avant.

Que nous apprend-il sur les propriétés de la table complète ? Il est évident qu'une *combinaison non unique* ne se transformera jamais en un identifiant potentiel pour la table complète.[4] Quant à l'ensemble des identifiants potentiels, on peut affirmer qu'il contient tous les identifiant potentiels de la table complète.

En d'autre termes, la table ID résultant de l'analyse partielle joue le rôle de la table COMB pour l'analyse complète. Après cette analyse partielle, on initialisera comme suit la table COMB, avant de procéder à l'analyse complète.

```
delete from COMB;
insert into COMB select * from ID;
```

La réduction du coût de l'analyse complète sera d'autant plus importante que la collection ID de l'analyse partielle sera réduite. On veillera donc à effectuer l'analyse partielle sur un sous-ensemble d'une taille significative pour minimiser le nombre d'identifiants potentiels non pertinents.

Un script complet

Le fichier **Recherche-identifiants.sql** contient un script commenté appliquant les principes développés dans cette section. Ce script effectue une pré-analyse des tables en calculant le nombre de lignes de chacune, le nombre de lignes distinctes, puis, pour chaque colonne, le nombre de valeurs distinctes et de lignes ayant une

3. On rappelle que le dictionnaire SQLfast n'est pas le catalogue de la base de données mais en est dérivé. On peut donc sans dommage en altérer le contenu, quitte à le restaurer par l'instruction `createDictionary`.

4. D'où le proverbe classique chez les rétro-ingénieurs : *Non unique un jour, non unique toujours*.

valeur *null*. L'utilisateur est ensuite invité à selectionner une série de tables dans lesquelles les identifiants potentiels sont recherchés. L'analyse de la base de données CLICOM.db donnerait ce qui suit :

```
--- Analyse de la base de données D:/SQLfast/CLICOM.db ---

La table CLIENT contient 16 ligne(s) dont 16 distincte(s)
   la colonne NCLI possède 16 valeur(s) distincte(s)
   la colonne NOM possède 15 valeur(s) distincte(s)
   la colonne ADRESSE possède 16 valeur(s) distincte(s)
   la colonne LOCALITE possède 7 valeur(s) distincte(s)
   la colonne CAT possède 4 valeur(s) distinctes et 2 valeur(s) NULL
   la colonne COMPTE possède 10 valeur(s) distincte(s)

La table PRODUIT contient 7 ligne(s) dont 7 distincte(s)
   la colonne NPRO possède 7 valeur(s) distincte(s)
   la colonne LIBELLE possède 7 valeur(s) distincte(s)
   la colonne PRIX possède 7 valeur(s) distincte(s)
   la colonne QSTOCK possède 7 valeur(s) distincte(s)

La table COMMANDE contient 7 ligne(s) dont 7 distincte(s)
   la colonne NCOM possède 7 valeur(s) distincte(s)
   la colonne NCLI possède 5 valeur(s) distincte(s)
   la colonne DATECOM possède 5 valeur(s) distincte(s)

La table DETAIL contient 14 ligne(s) dont 14 distincte(s)
   la colonne NCOM possède 7 valeur(s) distincte(s)
   la colonne NPRO possède 6 valeur(s) distincte(s)
   la colonne QCOM possède 13 valeur(s) distincte(s)

Table CLIENT
   41 combinaisons de 1 à 3 colonnes parmi 6 ont été générées.
   13 combinaisons ont été évaluées.
   6 identifiants potentiels ont été trouvés.

+------------------------+
| Identifiants de CLIENT |
+------------------------+
| ADRESSE                |
| NCLI                   |
| NOM,CAT                |
| NOM,COMPTE             |
| NOM,LOCALITE           |
| LOCALITE,CAT,COMPTE    |
+------------------------+

Table PRODUIT
   14 combinaisons de 1 à 3 colonnes parmi 4 ont été générées.
   4 combinaisons ont été évaluées.
   4 identifiants potentiels ont été trouvés.

+-------------------------+
| Identifiants de PRODUIT |
+-------------------------+
| LIBELLE                 |
| NPRO                    |
| PRIX                    |
| QSTOCK                  |
+-------------------------+
```

L'extension de ce script à l'exclusion des colonnes non pertinentes et à l'analyse préalable d'un échantillon réduit ne pose pas de problème et est laissée à l'initiative du lecteur.

A22.2ANALYSE DES VALEURS D'UNE COLONNE

Lorsque la nature des informations est inconnue ou mal connue, éventuellement de qualité incertaine, il est utile de procéder à une analyse de l'ensemble des valeurs des colonnes de la table concernée. On déterminera, notamment, pour chaque colonne (`$col$` est le nom de la colonne courante) :

- le nombre de valeurs (ou d'instances, doublons compris, *null* exclu)
  `count($col$)`
- le nombre de valeurs distinctes (*null* exclu
  `count(distinct $col$)`
- le nombre de valeurs numériques
  `sum(case when isNumeric($col$) then 1 else 0 end)`
- le nombre de valeurs non numériques
  `sum(case when isNotNumeric($col$) then 1 else 0 end)`
- le nombre de valeurs *null* ?
  `sum(case when $col$ is null then 1 else 0 end)`
- le nombre de valeurs vide
  `sum(case when $col$ = '' then 1 else 0 end)`
- la longueur minimale des valeurs (espaces exclus)
  `min(length(trim($col$)))`
- la longueur moyenne des valeurs (arrondie à une décimale, espaces exclus)
  `round(avg(length(trim($col$))),1)`
- la longueur minimale des valeurs (espaces exclus)
  `max(length(trim($col$)))`
- la valeur minimale de la colonne (limitée à une longueur maximale)
  `substr(cast(min($col$) as char),1,$limit$)`
- la valeur maximale de la colonne (limitée à une longueur maximale)
  `substr(cast(max($col$) as char),1,$limit$)`

Ces résultats permettent d'affiner le type de valeurs d'une colonne dont la définition est trop large, par exemple `varchar`, `char(1024)` ou `CLOB`. Le script A22.3 calcule ces métriques pour les colonnes des tables de la base de données CLICOM.db.

Le script complet est disponible sous la forme du fichier **Analyse-colonnes.sql**.

D'autres métriques permettent d'estimer la structure des valeurs numériques (entiers ou réels) et non numériques, dans certains cas à l'aide de jeux de données de référence externes fiables : dates, temps, numéro de téléphone, code postaux, coordonnées géographiques, adresses, noms géographiques, prénoms, patronymes, noms d'entreprises, codes NACE, etc.

```
openDB CLICOM.db;
createDictionary;
for tid,tab = [select TableID,TableName
               from   SYS_TABLE order by TableID];
  extract ncol = select count(*) from SYS_COLUMN
                 where TableID = $tid$
  set Q = ;
  set I = 0;
  for seq,col = [select ColSeq,ColName from SYS_COLUMN
                 where  TableID = $tid$ order by ColSeq];
    compute I = $I$ + 1;
    set Q = $Q$ select $seq$ as Seq,'$col$' as Colonne,;
    set Q = $Q$ count($col$) as "Nb inst",;
    set Q = $Q$ count(distinct $col$) as "Nb val",;
    ...
    set Q = $Q$ from $tab$;
    if ($I$ < $ncol$) set Q = $Q$ union;
  endfor;
set Q = $Q$ order by Seq;
$Q$;
```

Figure A22.3 - Analyse des valeurs des colonnes d'une base de données. La requête d'analyse est construite dans la variable **Q** puis exécutée.

L'estimation du type de codage est également très importante : codes 8 bits (ASCII, cp850, cp1252, Mac Roman) ou étendus (unicode). Elle se fera par le décodage selon les différents schémas de codage envisagés et l'observation du succès ou de l'échec de l'opération.

A22.3RECHERCHE DES CLÉS ÉTRANGÈRES D'UNE TABLE PAR ANALYSE DES DONNÉES

On considère dans la base de données à analyser un couple de tables (**A**,**B**). On recherche les combinaisons de colonnes de **B** susceptibles de constituer une clé étrangère implicite (non déclarée) vers **A**.

On fait l'hypothèse que les identifiants de **A** ont été détectés et qu'on a attribué aux colonnes des deux tables un type raisonnablement précis.
Suggestions à développer :

1. on part d'un identifiant **IA** de **A** pour lequel on recherchera dans **B** les clés étrangères potentielles
2. pour ce faire, on recherche dans **B** les combinaisons de colonnes **RB** dont le type est compatible avec celui des composants de **IA**
3. pour chaque combinaison **RB**, on vérifie la contrainte d'inclusion de ses valeurs dans celles de **IA**, par exemple en comptant les lignes de **B** dont la combinaison de valeurs de **RB** n'existe pas dans **A** comme combinaison de valeurs

de **IA**. Si la requête (symbolique)

```
select count(1) from B
where not exists(select 1 from A
                 where IA = B.RB);
```

renvoie `0`, la propriété d'inclusion est démontrée, ce qui fait de **RB** une clé étrangère potentielle. La probabilité que **RB** soit effectivement une clé étrangère est d'autant plus grande que le nombre de ligne de **B** est élevé.

4. Si le nombre de combinaisons à évaluer est important et si la table **B** est de grande taille, on évaluera d'abord la propriété d'inclusion sur un sous-ensemble de **B**. Si la propriété n'est pas invalidée, on l'évaluera sur un sous-ensemble plus grand, jusqu'à traiter la totalité des lignes de **B**.
5. On tiendra compte des clés étrangères cycliques (tables auto-référençantes).
6. En cas d'échec, on tiendra compte de la qualité des données. Le fait que certaines lignes de **B** violent la contraintes d'inclusion peut être dû à des données erronées.
7. On tiendra aussi compte en cas d'échec du fait que la contrainte référentielle peut ne s'appliquer qu'à un sous-ensemble des lignes de **B**.

Ce processus n'a pas encore fait l'objet d'une implémentation en SQLfast