Les sous-requêtes SQL scalaires de type "full select"

- Edito -

Date de mise en ligne : jeudi 12 novembre 2009

Environnement iSeries

Je vais vous parler aujourd'hui des sous-requêtes scalaires de type « full select », et vous présenter différentes façons de les utiliser.

Les sous-requêtes « full select » sont relativement simples à mettre en oeuvre, et se révèlent très pratiques dans de nombreux cas. Pourtant elles sont méconnues, et trop peu utilisées par la communauté des développeurs System i (tous langages de programmation confondus).

Les sous-requêtes scalaires de type « full select » se conforment à 2 principes qui sont les suivants :

1. elles ne peuvent renvoyer qu'une seule ligne,
2. elles ne peuvent renvoyer qu'une seule colonne.

Le 1er principe ne souffre aucune exception, en revanche le second principe peut faire l'objet d'exceptions dans certains cas particuliers que nous verrons plus loin.

Il est important de savoir qu'une sous-requête « full select » peut être utilisée dans différentes parties d'une requête SQL. Vous pouvez placer ces sous-requêtes dans la liste des colonnes de votre clause SELECT, mais aussi comme critère de sélection de votre clause WHERE, voire éventuellement comme source ou élément de jointure de votre clause FROM. Vous pouvez également utiliser plusieurs sous-requêtes « full select » à l'intérieur d'une même requête.

Les 2 exemples ci-dessous fonctionnent aussi bien avec DB2/400 (au moins à partir de la V5R3) qu'avec MySQL 5 :

Exemple 1 - dans la requête ci-dessous, on souhaite obtenir, pour chaque article du catalogue, son code, son libellé, le niveau de son stock d'alerte, et le stock réel de ce produit. Le stock d'alerte est défini dans la table « article », et le stock réel est situé dans la table « stock », le chaînage entre les deux se fait via la colonne « cod_art » qui a le même nom dans les 2 tables.

```
SELECT a.cod_art, a.lib_art, a.stk_alerte,
 (SELECT SUM(stock)
 FROM stock b
 WHERE b.cod_art = a.cod_art) AS stock_total
 From article a
```

Exemple 2 - en supposant que j'aie un identifiant dans une table et le libellé correspondant à cet identifiant dans une autre table, je peux afficher l'identifiant et son libellé sur la même ligne avec la requête suivante :

```
SELECT
 A.ID,
 (SELECT B.LIBELLE FROM TABLIB B WHERE A.ID = B.ID
 ) AS LIBELLE
 FROM TABCOD A
```

Cette technique peut être intéressante également si vous avez une table contenant des libellés différents pour chaque langue. Si l'on suppose que la table TABLIB a une clé composée d'un identifiant et d'un code langue, et si l'on souhaite afficher les libellés en français, on peut écrire ceci :

```
SELECT
 A.ID,
```

```
(SELECT B.LIBELLE FROM TABLIB B
WHERE A.ID = B.ID AND B.LANG = 'FR'
) AS LIBELLE
FROM TABCOD A
```

Cette technique peut être utilisée aussi bien dans les modes "SQL statique" que "SQL dynamique" de DB2/400. Elle peut donc être utilisée en RPG, en Adelia, en Cobol, en PHP, etc...

Au début de cet article, j'indiquais que vous pouviez utiliser cette technique également dans une clause WHERE ou dans une clause FROM, je vous laisse le soin d'explorer ces 2 possibilités par vous-même. Au niveau d'une clause WHERE, cela ne présente pas de difficulté particulière, au niveau d'une clause FROM, j'avoue que j'ai du mal à voir l'intérêt que cela peut présenter. Mais je n'ai pas la science infuse, en y réfléchissant vous trouverez probablement des cas d'utilisation intéressants pour vous.

On va rester au niveau de la clause SELECT avec un exemple légèrement plus complexe que le précédent. Pour ce nouvel exemple, je souhaite afficher une liste d'articles (avec code et libellé), avec en face de chaque article son prix de vente courant (calculé en fonction de la date du jour). Ce prix de vente est soumis à une date d'effet et est stocké dans une table annexe de la table article. Comme ça se corse, je vous propose de constituer un jeu d'essai, que voici :

```
CREATE TABLE ARTICLE (
 CODSOC CHAR ( 3)        NOT NULL WITH DEFAULT,
 CODART INTEGER          NOT NULL WITH DEFAULT,
 LIBELLE CHAR(30)        NOT NULL WITH DEFAULT,
 CONSTRAINT CLE_PRIM_ART PRIMARY KEY (CODSOC, CODART))

 INSERT INTO ARTICLE (CODSOC, CODART, LIBELLE) VALUES
 ('001', 1, 'article 1'),
 ('001', 2, 'article 2'),
 ('001', 3, 'article 3')

 CREATE TABLE PRXVTE (
 CODSOC CHAR ( 3)        NOT NULL WITH DEFAULT,
 CODART INTEGER          NOT NULL WITH DEFAULT,
 DATEFF DATE             NOT NULL WITH DEFAULT,
 PXVENT DECIMAL ( 11, 2) NOT NULL WITH DEFAULT,
 CONSTRAINT CLE_PRIM_PXV PRIMARY KEY (CODSOC, CODART, DATEFF))

 Constitution d'un jeu d'essai :
 INSERT INTO PRXVTE (CODSOC, CODART, DATEFF, PXVENT) VALUES
 ('001', 1, '2009-01-01', 1,5),
 ('001', 1, '2009-04-01', 2,2),
 ('001', 1, '2009-07-15', 2,5),
 ('001', 1, '2009-10-01', 3,1),
 ('001', 2, '2009-01-01', 4,1),
 ('001', 2, '2009-04-01', 4,3),
 ('001', 2, '2009-07-01', 5,1),
 ('001', 2, '2009-10-15', 4,9)
```

A noter : la colonne « COD_SOC » (pour « code société ») n'a pas une grande importance dans le cas qui nous préoccupe, mais j'ai trouvé intéressant de l'insérer comme élément de clé primaire, car les tables des applications de gestion sur plateforme System i sont très souvent constituées de la sorte.

J'imagine que certains petits malins vont vouloir utiliser la première technique SQL présentée dans l'article qui explique comment traiter des données soumises à date d'effet. D'ailleurs c'était mon premier réflexe à moi aussi. Adaptée au cas qui nous préoccupe, cela nous donne :

```
SELECT A.CODSOC, A.CODART, A.LIBELLE,
 ( SELECT B.PXVENT FROM PRXVTE B
 WHERE B.CODSOC = A.CODSOC
 AND B.CODART = A.CODART
 AND B.DATEFF <= CURRENT_DATE
 ORDER BY B.CODSOC, B.CODART, B.DATEFF DESC
 FETCH FIRST 1 ROW ONLY
 ) AS PRIX_VENTE
 FROM ARTICLE A
```

On a donc une requête SQL classique pour la récupération des codes et libellés d'articles, et on utilise une sous-requête scalaire de type « full select » pour la récupération du prix de vente.

C'est bien essayé, mais manque de chance, DB2 va vous envoyer dans les cordes avec un beau message d'erreur, en vous indiquant que la clause ORDER BY n'est pas supportée dans ce type de sous-requête. Et si vous enlevez la clause ORDER BY, vous aurez un message similaire sur la clause FETCH, qui elle non plus n'est pas supportée dans ce type de sous-requête.

N.B. avec MySQL5, la technique ci-dessus fonctionnerait sans problème, à condition de remplacer la clause FETCH FIRST 1 ROW ONLY par la clause LIMIT 1.

Puisque la première technique SQL ne fonctionne pas avec DB2, nous allons recourir à la seconde, ce qui donne à l'arrivée :

```
SELECT A.CODSOC, A.CODART, A.LIBELLE,
 ( SELECT B.PXVENT  FROM PRXVTE B
 WHERE B.CODSOC = A.CODSOC
 AND B.CODART = A.CODART
 AND B.DATEFF = (
 SELECT MAX(C.DATEFF) FROM PRXVTE C
 WHERE C.CODSOC = A.CODSOC
 AND C.CODART = A.CODART
 AND C.DATEFF <= CURRENT_DATE )
 ) AS PRIX_VENTE
 FROM ARTICLE A
```

Bon d'accord, à la première lecture, ça donne un peu mal à la tête, mais si vous creusez le truc, vous allez constater que c'est très logique. La sous-requête scalaire renvoie la date d'effet la plus proche de la date du jour, ce qui permet à la requête principale de récupérer le prix correspondant à cette date.

Lancée le 1er octobre 2009, en fonction du jeu d'essai constitué préalablement, la requête ci-dessus nous renvoie les informations suivantes :

```
CODSOC   CODART   LIBELLE                    PRIX_VENTE
 001          1   article 1                        3,10
 001          2   article 2                        5,10
 001          3   article 3                           -
```

On constate que le prix de l'article n° 3 est à NULL, ce qui est normal, compte tenu de notre jeu d'essai.

Si le fait de récupérer une valeur nulle vous embête, vous pouvez régler le problème en utilisant la fonction IFNULL, de la façon suivante :

```
SELECT A.CODSOC, A.CODART, A.LIBELLE,
 IFNULL(
 ( SELECT B.PXVENT FROM PRXVTE B
 WHERE B.CODSOC = A.CODSOC
 AND B.CODART = A.CODART
 AND B.DATEFF = (
 SELECT MAX(C.DATEFF) FROM PRXVTE C
 WHERE C.CODSOC = A.CODSOC
 AND C.CODART = A.CODART
 AND C.DATEFF <= CURRENT_DATE )
 ) , 0)  AS PRIX_VENTE
 FROM ARTICLE A
```

Et voilà le travail :

```
CODSOC   CODART   LIBELLE                    PRIX_VENTE
 001          1   article 1                        3,10
 001          2   article 2                        5,10
 001          3   article 3                        0,00
```

N.B. : l'exemple ci-dessus fonctionne aussi bien avec DB2/400 qu'avec MySQL 5.

Les exemples présentés jusqu'ici fonctionnaient aussi bien en SQL statique que dynamique. Du fait qu'elles fonctionnaient aussi en SQL dynamique, elles étaient utilisables avec PHP et les langages apparentés.

Les techniques que nous allons voir maintenant sont spécifiques au mode "SQL statique" de DB2/400, elles ne peuvent donc être utilisées que dans des programmes compilés, écrits en RPG/400 ou en Cobol/400. Elles fonctionnent également très bien en Adelia, pour ceux qui utilisent cet AGL. Ces techniques sont très (trop) rarement utilisées en entreprise, et c'est bien dommage car je trouve qu'elles apportent beaucoup de souplesse dans le processus de développement (je reviendrai sur certains de leurs avantages à la fin de l'article).

Les exemples ci-dessous ont pour particularité de combiner l'utilisation des sous-requêtes scalaires de type « full select » avec les syntaxes suivantes :

- SELECT ... INTO :variables_hôte(s) FROM ....
- SET :variable_hôte = (sous requête)
- VALUES (sous-requête) INTO :variables_hôte(s)

Voyons tout de suite différents exemples :

La requête SQL statique ci-dessous consiste à lire une ligne unique de la table T1, et à renvoyer le contenu d'une colonne C2 dans une variable hôte V2 :

```
 V2 = *BLANK    * ou V2 = 0 si numérique
 DEBUT_SQL
```

```
+ SELECT C2
+   INTO :V2
+ FROM T1
+ WHERE C1 = :V1
FIN_SQL
```

A noter : l'exemple ci-dessus est donné dans la syntaxe Adelia. Pour écrire l'équivalent en RPG, il vous suffit de remplacer DEBUT_SQL par Exec SQL et FIN_SQL par End-Exec. Nous verrons plus loin des exemples écrits en syntaxe purement RPG.

En fait la forme d'écriture ci-dessus, basée sur l'association des prédicats SELECT et INTO, peut s'écrire sous 2 autres formes :

- 1ère forme : avec la clause SET

```
V2 = *BLANK    * ou V2 = 0 si numérique
DEBUT_SQL
+ SET :V2 = (SELECT C2 FROM T1 WHERE C1 = :V1)
FIN_SQL
```

- 2ème forme : avec la clause VALUES ... INTO

```
V2 = *BLANK    * ou V2 = 0 si numérique
DEBUT_SQL
+ VALUES (SELECT C2 FROM T1 WHERE C1 = :V1) INTO :V2
FIN_SQL
```

IMPORTANT : On va maintenant introduire une exception à l'un des principes vus précédemment. Si on ne peut en aucun cas faire exception au fait qu'une sous-requête « full select » ne renvoie qu'une seule ligne, on verra dans l'exemple qui suit que l'on peut dans certains cas renvoyer plusieurs colonnes.

Avec SELECT INTO, le développeur a la possibilité de récupérer plusieurs valeurs en sortie d'une requête « full select ». Il a pour ce faire plusieurs techniques à sa disposition :

- première forme « traditionnelle » :

```
V4 = *BLANK    * ou V4 = 0 si numérique
V5 = *BLANK    * ou V5 = 0 si numérique
DEBUT_SQL
+ SELECT C4, C5
+   INTO :V4, :V5
+ FROM T1
+ WHERE C1 = :V1 AND C2 = :V2
FIN_SQL
```

- seconde forme utilisant la clause VALUES ... INTO :

```
V4 = *BLANK    * ou V4 = 0 si numérique
V5 = *BLANK    * ou V5 = 0 si numérique
DEBUT_SQL
+ VALUES (SELECT C4, C5 FROM T1
+   WHERE C1 = :V1 AND C2 = :V2)
```

```
+   INTO :V4 , :V5
FIN_SQL
```

Autre exemple démontrant la facilité de manipulation des dates, cette fois-ci dans la syntaxe SQL RPG :

```
H DECEDIT('0,') DATEDIT(*YMD)  DEBUG
 *
 DWVARO1           S               d
 DWVARO2           S               d
 DWVARI1           S              2s 0 inz(1)
 DWVARI2           S              2s 0 inz(3)
 *
 C/Exec SQL
 C+ VALUES (SELECT CURRENT DATE,
 C+  CURRENT DATE + :WVARI1 MONTHS - :WVARI2 DAYS
 C+  FROM QSQPTABL)
 C+ INTO :WVARO1 , :WVARO2
 C/End-Exec
 C                   DUMP
 C                   SETON                                        LR
```

Si l'on exécute ce code le 22/06/2009, le Dump produit le résultat suivant :

```
 WVARI1               ZONED(2,0)          01.
 WVARI2               ZONED(2,0)          03.
 WVARO1               DATE(10)            '2009-06-22'
 WVARO2               DATE(10)            '2009-07-19'
```

ATTENTION : On rappelle que ces différentes techniques ne fonctionnent qu'à la seule condition que les sous-requêtes considérées ne renvoient qu'une seule ligne (principe du « full select »). Si la sélection définie dans la clause WHERE ne permet pas de récupérer à coup sûr une seule ligne, il est possible dans certains cas de contourner le problème en ajoutant à la requête la clause « FETCH FIRST 1 ROW ONLY » comme dans l'exemple ci-dessous :

```
V2 = *BLANK    * ou V2 = 0 si numérique
DEBUT_SQL
+ SELECT C2
+   INTO :V2
+ FROM T1
+ WHERE C1 = :V1
+ FETCH FIRST 1 ROW ONLY
FIN_SQL
```

La technique présentée ci-dessus peut être utilisée également pour réaliser des comptages (ou encore des sommes ou des moyennes), à la condition de ne renvoyer, là-encore, qu'un seul enregistrement en sortie (l'utilisation de la clause GROUP BY est à proscrire dans ce cas). Exemple ci-dessous avec la fonction de comptage COUNT :

```
*-- Initialisation de la variable de comptage
VN_NBR = 0
*-- Requête de comptage
DEBUT_SQL
+ SELECT COUNT(*)
+   INTO :VN_NBR
+ FROM T1
```

```
+ WHERE C1 = :V1
FIN_SQL
*
SI *SQLCODE >= 0 ET *SQLCODE <> 100
SI VN_NBR > 0
... traitement fonctionnel ...
FIN
FIN
```

Une autre application pratique des requêtes de type « full select » consiste à les utiliser dans les algorithmes de contrôle de validité de zone, et de récupération de libellés. Par exemple, dans le pavé VERIFICATION d'un programme Adelia, au lieu d'écrire ceci :

```
SI ZZZ_NUM_CLI = *BLANK
* Zone obligatoire
ZZZ_LIB = *BLANK
PREPARER_MSG 0001 ZZZ_NUM_CLI
ANOMALIE
SINON
LIRE TABCLI
SI TABCLI EXISTE
ZZZ_LIB = LIBCLI
SINON
* Code erroné, non trouvé dans la table TABCLI
ZZZ_LIB = *BLANK
PREPARER_MSG 0002 ZZZ_NUM_CLI
ANOMALIE
FIN
FIN
```

On peut écrire ceci :

```
SI ZZZ_NUM_CLI = *BLANK
* Contrôle de type « Zone obligatoire »
ZZZ_LIB = *BLANK
PREPARER_MSG 0001 ZZZ_NUM_CLI
ANOMALIE
SINON
* Initialisation de l'indicateur de recherche SQL
W_SQL_TROUVE = *BLANK
* Initialisation du libellé affiché à l'écran
ZZZ_LIB = *BLANK
DEBUT_SQL
+ VALUES (SELECT '1', LIBCLI FROM TABCLI WHERE
+   CODSOC = :ZZZ_COD_SOC AND CODCLI = :ZZZ_NUM_CLI)
+   INTO :W_SQL_TROUVE , :ZZZ_LIB
FIN_SQL
SI W_SQL_TROUVE <> '1' OU *SQLCODE <> 0
* Code erroné, non trouvé dans la table TABCLI
PREPARER_MSG 0002 ZZZ_NUM_CLI
ANOMALIE
FIN
FIN
```

CONCLUSION : les techniques présentées ici offrent à mon avis deux avantages majeurs :

- elles permettent de s'affranchir des problèmes de niveau de version, inhérents à l'utilisation d'ordres de lecture natifs. Ainsi, si la structure de la table TABCLI est modifiée, alors les programmes qui accèdent à cette table via SQL - simplement pour contrôler un identifiant et/ou récupérer un libellé - n'ont plus besoin d'être recompilés. On réduit ainsi le nombre d' « adhérences » entre les programmes et les fichiers DB2, et on gagne en réactivité au niveau du processus de développement. Bien évidemment, ce que je viens d'écrire n'est valable que dans les cas où la modification de la table TABCLI ne concerne pas les colonnes utilisées dans la sous-requête scalaire.

- même si la table SQL utilisée dans la sous-requête contient beaucoup de colonnes et possède un gros buffer, le programme qui utilise cette sous-requête ne « voit » que les colonnes dont il a réellement besoin. Cela permet de garantir d'excellentes performances (à condition bien évidemment, que la table utilisée dans la sous-requête possède un index adapté à la sélection demandée).

