LES REQUETES SQL

Quelques définitions et beaucoup d'exemples

.1 Définitions et généralités

SQL (Structured Query Language (langage d'interrogation structuré) est un langage d'interrogation pour les bases de données. Il permet de récupérer les enregistrements d'une base de données correspondant à certains critères sans que le programmeur n'ait à se soucier des index et des relations entre les tables. Le langage a été étendu pour inclure des commandes de mises à jour des bases de données et des commandes 'système'. VFP inclus un **SUR**ensemble du langage SQL : VFP permet des choses non admises par la norme, en particulier l'utilisation de simples tables libres comme sources de données ainsi que l'utilisation des fonctions VFP (et celles définies par l'utilisateur). Dans ce document nous essaierons de signaler les éléments 'hors norme'.

Il existe, dans VFP, un générateur de requête SQL (en net progrès sur la version 9) mais il est assez limité surtout lorsque les requêtes sont importantes. Avec un peu d'habitude, il devient vite plus rapide d'écrire la requête à la main même si, par exemple, on se sert du générateur pour obtenir la liste des champs. Ce document ne parlera pas ou peu de ce générateur ; une fois les clauses SQL comprises, l'utilisation du générateur est immédiate.

Une **vue** est un moyen d'accéder à (et/ou de modifier) des enregistrements d'une table (libre ou incluse dans une base de données) au moyen d'une commande SQL qui peut être paramétrée. Utiliser une vue est très facile quand on connaît SQL. IL est beaucoup plus difficile de réfléchir aux problèmes des mises à jour concurrentielles et de définir les actions à executer en cas de conflit. Une description complète devrait vous être faite dans le compte rendu de la session C6 des rencontres de La défense (novembre 2004).

L'exécution d'une requête SQL crée, dans les cas les plus courant, soit un curseur a priori en lecture seule (ce curseur peut, normalement, être indexé mais pas modifié (avant VFP7 il y a un moyen de 'truander', à partir de VFP7 il existe une clause READWRITE qui autorise la modification du curseur) soit une table 'standard'. Dans la documentation en anglais, le mot CURSOR recouvre les deux types : table réelle et curseur 'en mémoire' (ou en .tmp). Un curseur s'utilise comme une table (à la restriction indiquée ci-dessus) ; il disparaît 'physiquement' (de la mémoire ou du disque) dès qu'on le ferme.

Dans ce document, nous allons essayer de donner le maximum d'exemples dans lesquels vous pourrez puiser des idées. C'est une version 0. Je vous demande de m'envoyer vos 'plus belles requêtes' pour que je puisse le compléter et le rendre ainsi encore plus utile. Je vous demande aussi de le critiquer : quels sont les points obscurs, les choses mal dites, les erreurs, etc …

.2 requêtes SQL

Nous allons regarder les principales clauses de la commande SQL **SELECT.** Cette commande comprend :

1- la liste des champs à récupérer ; une étoile '*' indique tous les champs (il y a une petite perte de performance à indiquer * à la place de la liste des champs). Un certain nombre de fonctions font parties du langage : SUM(), AVG(), COUNT(), …. ; VFP autorise aussi l'utilisation des fonctions utilisateurs (avec une perte de performance très sensible) ;
2- la liste des tables (dans VFP : vues ou curseurs ou tables libres ou incluses dans une base de données) ;
3- les relations entre les tables (relations locales à la commande SQL) ; ces relations locales sont appelées 'jointures' ;
4- les caractéristiques des enregistrements (lignes) de chaque table récupérés pour créer le résultat (filtre) ;
5- les critères de regroupement (en général pour les fonctions d'agrégation (de calcul)) ;
6- l'ordre dans lequel doit se trouver le résultat ;
7- la nature du résultat (tableau de variables, curseur, table).

Dans les paragraphes suivants, nous prenons comme support la base de données TESTADA dans les exemples de VFP (les exemples fonctionneraient à l'identique sur des tables libres). Dans certaines version de VFP cet exemple est dans TASTRADE et le nom des champs est légèrement modifié (customer_id au lieu de cust_id par exemple).

.2.1. requêtes mono-table

création d'un curseur contenant tous les clients français :

```
SELECT * FROM customer WHERE country="France" INTO CURSOR france
```

NOTE : le = n'obéit pas au SET EXACT mais au SET ANSI ; voir plus loin la clause LIKE

les mêmes mais classés par ordre de maxordamt décroissant (la plus grande valeur est dans le premier enregistrement) et dans une table; notez que l'on part du curseur précédent et non de la table origine ; notez aussi que la table créée ne fait pas partie de la base de données (c'est une table libre) :

```
SELECT * FROM france ORDER BY maxordamt DESCENDING INTO DBF frmax
```

on ne veut que les 5 premiers (notez qu'en VFP le ' ;' sert de caractère de continuation. C'est l'inverse dans la norme (il sert à terminer la commande)) :

```
SELECT TOP 5 * FROM customer WHERE country="France" ORDER BY maxordamt ;
 DESCENDING INTO CURSOR frmax5
```

NOTE : la clause TOP est trompeuse (surtout en réseau) : le SQL commence par récupérer tous les enregistrements puis en dernier lieu ne donne que ce qu'il faut.

calcul du nombre de clients par pays (notez que l'on trouve ainsi un client sans pays) :

```
SELECT country, COUNT(*) AS nombre FROM customer ;
 GROUP BY 1 ORDER BY 1 INTO CURSOR pays
```

NOTE : lorque la clause ORDER est strictement identique à la clause GROUP BY, elle n'est pas obligatoire.
NOTE : cette requête est strictement aux normes.

calcul du nombre de clients par pays mais, en plus, on voudrait avoir un des clients de chaque pays :

```
SELECT country, COUNT(*) AS nombre, customer_id, company_name, ;
 contact_name, contact_title, address, city, region, postal_code ;
FROM customer ;
 GROUP BY 1 ORDER BY 1 INTO CURSOR pays
```

NOTE : cette requête qui fonctionne parfaitement en VFP avant la version 9 n'est pas aux normes car dans la liste des champs il y en a qui n'apparaissent pas dans la clause GROUP BY. Si vous voulez travailler avec une base distante (non VFP) il faut corriger la requête. Si vous travaillez avec VFP9, vous pouvez utiliser la commande SET ENGINEBEHAVIOR 70 pour que VFP fonctionne comme dans les versions précédentes.

liste des pays ayant plus de 3 clients :

```
SELECT country, COUNT(*) AS nombre FROM customer ;
  GROUP BY 1 HAVING COUNT(*) > 3 INTO CURSOR pays3
```

calcul de la moyenne de maxordamt (on obtient un seul enregistrement car il n'y a pas de clause GROUP BY) :

```
SELECT AVG(maxordamt) AS moyenne FROM customer INTO CURSOR moyenne
```

Une requête peut admettre une sous-requête (VFP6 et avant) ou plusieurs : on veut tous les clients dont la maxordamt est supérieur à la moyenne :

```
SELECT * FROM customer ;
  WHERE maxordamt >= ANY (SELECT moyenne FROM moyenne) ;
    INTO CURSOR riches
```

NOTE : ici la clause ANY peut prêter à confusion puisque moyenne ne contient qu'un seul enregistrement mais VFP s'y retrouve !

on veut les clients français sans commande :

```
SELECT * FROM customer WHERE country="France" ;
  AND cust_id NOT IN (SELECT cust_id FROM orders) INTO CURSOR cmd0
```

NOTE : dans le chapitre suivant vous trouverez la même requête mais sous une autre forme
NOTE : jusqu'à VFP8 on ne peut mettre qu'une seule sous-requête. A partir de VFP9, on peut en mettre plusieurs.

Mise en évidence du danger d'utiliser une fonction utilisateur dans une requête SQL. Créons une petite fonction que l'on enregistre dans cpv.prg :

```
FUNCTION cpv
  LPARAMETERS cp, ville
RETURN ALLTRIM(cp)+ ' '+ ALLTRIM(ville)
```

On fait un SET PROCEDURE TO cpv et on écrit la requête :

```
SELECT cpv(postal_code,city) AS cpv ;
  FROM customer WHERE country="EU" INTO CURSOR pb1
```

Vous allez voir que la plupart des noms de ville est tronquée dans le résultat : SQL ne pouvant déterminer la taille du champ cpv dans le résultat a commencé par lancer la fonction avec la première ligne qui lui tombait sous la main puis il a récupéré le résultat et sa longueur est devenue la taille du champ. Imaginer ce qu'il peut se passer si la taille est très petite ! ! !

.2.2. jointure entre tables (relations locales) : clause JOIN

Une jointure est une relation locale entre deux tables utilisée par SQL. Le résultat comprendra une ligne pour chaque couple 'parents-enfants'. Il arrive TRES souvent que suite à une jointure mal écrite ou à des problèmes de clef que le résultat soit le double de ce que l'on attendait parce que SQL a créé une ligne pour un sens de la relation et une autre pour l'autre sens. Il arrive aussi que l'on obtienne un résultat gigantesque parce que les clefs ne sont uniques ; c'est le cas des enregistrements vides : si vous avez 100 enregistrements vides dans une tables et 100 autres dans une autre, une jointure donnera 10000 lignes vides ….. En SQL, on peut relier une table avec elle-même (voir plus loin) ce qui permet des calculs intéressants.

Il y a deux 'écritures' pour les clauses JOIN : celle du générateur de vues qui les 'imbrique' mais cela devient vite illisible et une écriture 'séquentielle' qui me paraît beaucoup plus compréhensible.

Avant VFP6, la clause JOIN n'était pas supportée et les jointures étaient définies dans la clause WHERE avec un gros défaut : les jointures externes n'étaient pas possibles.

on veut les commandes des clients français :

```
SELECT customer.*, orders.* FROM customer ;
  INNER JOIN orders ON customer.cust_id = orders.cust_id ;
    WHERE customer.country="France" INTO CURSOR cmdclfr
```

NOTE : regardez l'utilisation des alias et de l'étoile dans le SELECT : cette écriture est très facile mais elle pose un problème : si deux champs ont le même nom dans les deux tables (par exemple ici cust_id), ils seront renommés dans le résultat en cust_ida et cust_idb et si vous vous attendez à trouver cust_id, il y aura une erreur.

la commande ci-dessus NE DONNE PAS les clients français qui n'aurait pas de commande. Si on les veut, il faire une jointure externe :

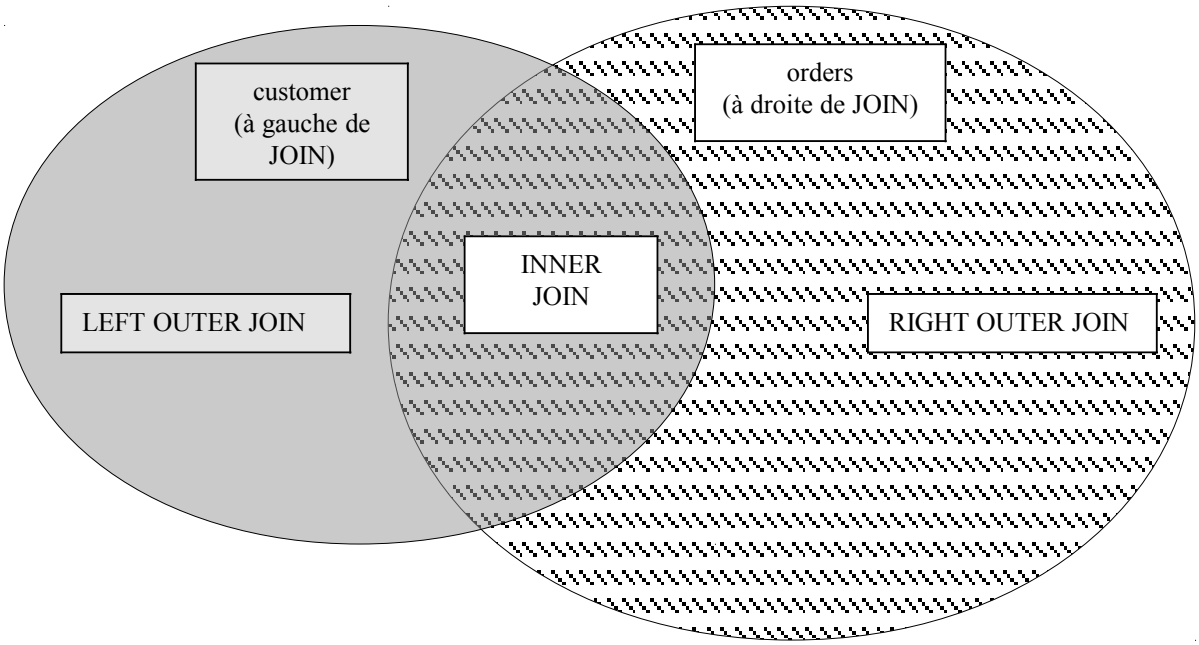
```
SELECT customer.*, orders.* FROM customer ;
  LEFT OUTER JOIN orders ON customer.cust_id = orders.cust_id ;
    WHERE customer.country="France" ORDER BY customer.cust_id ;
      INTO CURSOR clcmd2
```

Vous pouvez remarquer que le client de code « PARIS » a été ajouté, tous les champs provenant de orders contiennent .NULL. La présence de ces champs peut être testée avec la fonction ISNULL() mais bien souvent la présence de ces NULL pose problème. La fonction NVL() permet de remplacer un NULL par une valeur prédéfinie (par exemple une chaîne vide). Exemple

```
SELECT e.*, NVL(b.txtcantine,"") AS txtcantine, NVL(b.comment,"") AS comment ;
        FROM enfants e LEFT OUTER JOIN enfant2 b ON e.codfam+e.enfant = b.codfam+b.enfant;
         WHERE ……..
```

Le schéma ci-dessous explicite les résultats obtenus en utilisant

1. INNER JOIN : les lignes mise dans le résultat obéissent aux filtres sur les 2 tables
2. LEFT OUTER : on obtient toutes les lignes obéissant aux filtres sur les 2 tables plus toutes les lignes de la table placée à gauche de JOIN obéissant aux filtres (clause WHERE) sur cette table sans que la relation locale (le JOIN) ne trouve de ligne correspondante dans la table à droite de la clause JOIN.
3. RIGHT OUTER : comme ci-dessus mais en inversant la droite et la gauche.



la même commande avec des alias locaux (connus qu'à l'intérieur de la commande SQL). L'utilisation d'alias locaux simplifie énormément l'écriture (et la taille) des requêtes SQL (a priori les alias locaux ne sont pas gérés par le générateur de requêtes).

```
SELECT c.*, o.* FROM customer c ;
  LEFT OUTER JOIN orders o ON c.cust_id = o.cust_id ;
    WHERE c.country="France" ORDER BY c.cust_id   INTO CURSOR clcmd2
```

on reveut les clients français sans commande :

```
SELECT c.*, COUNT(*) AS nbcom FROM customer c ;
  LEFT OUTER JOIN orders o ON c.customer_id = o.customer_id ;
    WHERE c.country="France" ;
      GROUP BY o.customer_id HAVING COUNT(o.order_id)=0    INTO CURSOR clifr0c
```

Cette requête ne fonctionne pas ! Tout simplement parce qu'on demande le nombre de ligne du résultat (COUNT(*)) et que le LEFT OUTER JOIN nous donne une ligne même s'il n'y a pas de commande !

écrivons :

```
SELECT c.*, COUNT(o.order_id) AS nbcom FROM customer c ;
  LEFT OUTER JOIN orders o ON c.customer_id = o.customer_id ;
    WHERE c.country="France" ;
      GROUP BY o.customer_id HAVING COUNT(o.order_id)=0   INTO CURSOR clifr0c
```

on obtient bien une ligne puis que maintenant on compte le nombre de commandes et non pas le nombre de lignes dans le résultat !

ces deux exemples montrent qu'il faut faire EXTREMEMENT attention à ce que l'on écrit et qu'une petite faute, très difficilement perceptible, donnera des résultats totalement erronés.

Les deux écritures possibles pour les jointures multiples sont :

```
SELECT c.*, o.*, l.* FROM customer c;
 INNER JOIN orders o ON c.customer_id = o.customer_id ;
  INNER JOIN order_line_items l ON o.order_id = l.order_id ;
   WHERE c.country="France" INTO CURSOR cmde

SELECT * FROM customer INNER JOIN orders;
   INNER JOIN order_line_items ;
  ON  Orders.order_id = Order_line_items.order_id ;
  ON  Customer.customer_id = Orders.customer_id;
 WHERE Customer.phone = "France" INTO CURSOR cmde1
```

Certains mettent même les jointures entre parenthèses ! !

.2.3. clause UNION

La clause UNION vous permet de 'concaténer' plusieurs requêtes SQL ayant la même structure. Veuillez faire attention, dans l'exemple qui suit à la place des différentes clauses (un seul group by, un seul INTO, …). Normalement, la clause UNION supprime les lignes identiques (les doublons) sauf si on écrit UNION ALL. Les tables origines peuvent être différentes, le seul point important est que les différentes requêtes réunies par UNION aient **STRICTEMENT** la même structure. Une requête peut contenir plusieurs UNION.

```
SELECT c.* FROM customer c WHERE c.country = "France" ;
 INTO CURSOR UUU ;
UNION ALL ;
 SELECT u.* FROM customer u WHERE u.country = "EU" ;
  ORDER BY 2
```

.2.4. sacré Anders !

Lors de la réunion de Prague (Tchéquie 2003), Anders ALTBERG nous a montré une requête sur deux tables **sans jointure**. Dans ce cas, le résultat comporte une ligne pour chaque couple d'enregistrements. Soit la table JOUR contenant les jours du mois et une table heure contenant les heures d'une journée, la requête vous fournira …. un agenda !

.3 sous-requêtes

Les sous-requêtes, que l'on va pouvoir trouver à la place d'un champ, d'une clause FROM, d'une source d'un JOIN ou dans une clause WHERE, vont nous permettre de régler élégament et rapidement des problèmes ardus. Ce paragraphe n'est qu'une introduction, avec beaucoup d'exemples, de ce que l'on peut faire. Pour plus de détails, vous reporter au livre de tamar GRANOR "domptez le SQL" (j'ai participé à sa traduction : la pub est donc gratuite !)

.3.1. sous-requêtes à la place d'un champ

Dans mon logiciel de gestion d'école (www.babazou.fr), je dois retrouver, pour chaque élève, sa première année et sa dernière année de présence : on peut écrire :

```
CREATE SQL VIEW LISTE_ENFANTS_1FAMILLE_PREMIER_DERNIER AS ;
 SELECT DISTINCT codfam, enfant, datenai, sexe, nomenf, prenom, dat1entre, ;
    (SELECT MIN(f.ascol) FROM enfants f WHERE f.codfam=?lccodcou AND f.enfant=e.enfant ;
      AND f.datmodif={}) AS ascol1, ;
    (SELECT MAX(g.ascol) FROM enfants g WHERE g.codfam=?lccodcou AND g.enfant=e.enfant ;
      AND g.datmodif={}) AS ascold ;
 FROM enfants e ;
 WHERE codfam = ?lccodcou AND datmodif = {} ;
 ORDER BY datenai
```

Autre exemple :

Dans la table des règlements, il peut y avoir plusieurs lignes pour un seul règlement si celui-ci concerne plusieurs factures ou items. Les lignes d'un même règlement ont la même clef (il y a une sous-clef !). On veut le montant de chaque règlement quelque soit le nombre de ses lignes :

```
SELECT rr.clef, rr.codfam, ;
 (SELECT SUM(ss.montant) FROM paiement ss WHERE ss.clef=rr.clef) AS mtotal ;
FROM paiement rr ;
WHERE rr.clef>3000 ;
INTO CURSOR test1
```

Note : remarquez l'absence de la clause GROUP BY dans la sous-requête.
il n'est pas possible de mettre un filtre (clause WHERE) sur la colonne mtotal

.3.2. sous-requête dans la clause FROM

Nous avons vu plus haut que la norme SQL (en VFP : SET ENGINEBEHAVIOR 90) interdisait de mettre des champs qui ne participent pas à la clause GROUP BY dans la liste des champs. Un moyen de contourner ce problème est d'utiliser une sous-requête dans la clause FROM. Cela permet aussi de mettre un filtre (clause WHERE) sur le champ calculé (voir requête ci-dessus) :

```
SELECT t.clef, t.codfam, t.mtotal ;
FROM ;
   (SELECT r.clef, r.codfam, ;
      (SELECT SUM(s.montant) FROM paiement s WHERE s.clef=r.clef) AS mtotal ;
   FROM paiement r) t ;
WHERE mtotal BETWEEN 0 AND 100 ;
INTO CURSOR sss
```

Note : la requête ci-dessous est une autre manière de régler le même problème.

.3.3. sous-requête dans la clause JOIN

dans mon logiciel de gestion d'école, un règlement peut contenir plusieurs lignes (s'il correspond à plusieurs factures). On veut le montant total du règlement à coté du montant de chaque ligne :

```
SELECT p.clef, p.montant, p.modepaim, s.valeur ;
FROM paiement p ;
INNER JOIN ;
   (SELECT clef, SUM(montant) AS valeur FROM paiement ;
    WHERE clef>0 AND datmodif={} AND datannul={} ;
    GROUP BY 1) AS s ;
  ON p.clef = s.clef ;
INTO CURSOR testval
```

Note: une clef négative ou le champ datmodif (ou datannul) non vide indique une ligne non valide à ne pas prendre en compte.

cette requête est à peu près équivalente à celle décrite en 3.2

.4 quelques beaux exemples !

```
lcvue = "cycles_periodes_classesth_sections_suivante"
CREATE SQL VIEW (lcvue) ;
  AS SELECT c.cycle, c.libcycle, c.bits AS cyclebits, c.ordraff AS cycleordre, c.be1dnom4, ;
   p.periode AS grdperiode, p.libperiode AS periode, p.abrev AS grdabrev, ;
    p.bits AS grdbits, p.ordraff AS grdordre, ;
    q.periode AS cperiode, q.libperiode AS annees, q.abrev, q.bits, q.ordraff, ;
     t.classeth, t.classe, t.bits AS clabits, t.suivante AS csuivante, t.ordraff AS claordre, ;
      NVL(r.niveau + r.classeth, CHR(1)+CHR(1)+CHR(1)+CHR(1)) AS nc, ;
       NVL(s.niveau, CHR(1)+CHR(1)) AS cniveau, NVL(s.libelle, SPACE(15)) AS niveau, ;
        NVL(s.bits, CAST(1 AS INTEGER)) AS nivbits, ;
         NVL(s.ordraff, CAST(9999 AS INTEGER)) AS nivordre, ;
          NVL(s.be1dnom9, CAST(0 AS INTEGER)) AS be1dnom9, ;
           NVL(v.classe, SPACE(15)) AS suivante, NVL(v.bits, CAST(1 AS INTEGER)) AS suivbits ;
  FROM elv!periodes_scolaires p INNER JOIN elv!periodes_scolaires q ;
   ON LEFT(p.periode,1) = LEFT(q.periode,1) ;
    INNER JOIN elv!classes_theoriques t ON t.periode = q.periode ;
     INNER JOIN elv!cycles_scolaires c ON t.cycle = c.cycle ;
      LEFT OUTER JOIN elv!classes_niveaux r ON t.classeth = r.classeth ;
       LEFT OUTER JOIN elv!niveau_scolaire s ON s.niveau = r.niveau ;
        LEFT OUTER JOIN elv!classes_theoriques v ON t.suivante = v.classeth ;
  WHERE RIGHT(p.periode,1)= CHR(1) AND LEFT(p.periode,1) >= CHR(10) ;
   AND RIGHT(q.periode,1)>= CHR(10) ;
  ORDER BY c.ordraff, p.ordraff, q.ordraff, t.ordraff, nivordre


lcvue = "LISTE_ENFANT_1_FAMILLE_2DERNRAN"
CREATE SQL VIEW (lcvue) AS ;
 SELECT codfam, enfant, datenai, ascol, nomenf, prenom, sexe, classe, ;
  classex, RECNO() AS numenreg ;
  FROM enfants WHERE codfam = ?lccodcou AND ascol = ?lnannee AND datmodif = {} ;
UNION ;
 SELECT codfam, enfant, datenai, ascol, nomenf, prenom, sexe, classe, ;
  classex, RECNO() AS numenreg ;
 FROM enfants WHERE codfam = ?lccodcou AND ascol+1 = ?lnannee AND datmodif = {} AND ;
  enfant NOT IN (SELECT c.enfant FROM enfants c WHERE c.codfam = ?lccodcou AND ;
  c.ascol = ?lnannee AND c.datmodif={}) ;
ORDER BY 3


  lcvue = "LISTE_ENFANT_1_FAMILLE_TOUTAN"
  CREATE SQL VIEW (lcvue) ;
   AS SELECT codfam, enfant, datenai, ascol, nomenf, prenom, sexe FROM enfants ;
    WHERE codfam = ?lccodcou AND ascol = ?lnannee AND datmodif = {} ;
    UNION ;
    SELECT codfam, enfant, datenai, MAX(ascol) AS ascol, nomenf, prenom, sexe FROM enfants ;
    WHERE codfam = ?lccodcou AND datmodif = {} AND ;
     enfant NOT IN (SELECT c.enfant FROM enfants c ;
                 WHERE c.codfam = ?lccodcou AND c.ascol = ?lnannee ;
                 AND c.datmodif={}) ;
     GROUP BY 1, 2 ORDER BY 3
```

```
lcvue = "NOUVEAUX_ELEVES_VRAIS"
CREATE SQL VIEW (lcvue) AS SELECT e.*, NVL(b.txtcantine,SPACE(50)) AS txtcantine, ;
  NVL(b.comment, CAST("" AS M)) AS comment, NVL(b.ine, CAST("" AS C(11))) AS ine, ;
  NVL(b.medical, CAST("" AS M)) AS medical, NVL(b.dmedical, {}) AS dmedical, ;
  NVL(b.datenais, {}) AS datenais, NVL(b.inseenai, "     ") AS inseenai, ;
  NVL(b.communai, SPACE(20)) AS communai, NVL(b.bits21, CAST(1 AS Integer)) AS bits21, ;
  NVL(b.d1entre, {}) AS d1entre ;
FROM enfants e LEFT OUTER JOIN enfant2 b ON e.codfam+e.enfant = b.codfam+b.enfant;
WHERE e.ascol = ?lnannee AND e.datmodif = {} AND e.datannul = {} AND ;
   (BITTEST(e.bits,3)=.T. OR BITTEST(e.bits,5)=.T.) AND ;
   e.codfam+e.enfant NOT IN (SELECT codfam+enfant FROM enfants f WHERE f.ascol = ?lnannee-1 ;
   AND f.datmodif = {}) ;
ORDER BY e.classe, e.nomenf, e.prenom, e.codfam
```

La requête la plus complexe que j'ai faite : elle me permet de rechercher les réglements en fonction d'un certain nombre de critère. La difficulté vient du fait qu'un règlement peut avoir plusieurs lignes et que si on filtre sur le montant, il faut le faire soit sur le montant de la ligne soit sur le total des lignes d'un règlement. L'execution de cette vue sur une table de 38000 lignes prend 0,160s !

```
lcvue = "recherche_reglements"
CREATE SQL VIEW (lcvue) AS SELECT ;
  rr.codfam, rr.codfactur, rr.anscol, rr.trim, rr.montant, rr.pairemb, rr.type, rr.modepaim, rr.datremise, ;
  rr.differe, rr.datcreat, rr.datmodif, rr.datannul, rr.devise, rr.drgl, rr.secondes, rr.typrgl, ;
  rr.exprj, rr.souclef, rr.ltrfisc, rr.clef, rr.codbanque, rr.tireur, rr.numchq, rr.bits1, rr.cleffac, ;
  rr.comment, rr.endoss, rr.quiendos, rr.souclemax, rr.cleprlvaut, mtotal ;
FROM ;
    (SELECT tt.codfam, tt.codfactur, tt.anscol, tt.trim, tt.montant, tt.pairemb, tt.type, ;
      tt.modepaim, tt.datremise, ;
      tt.differe, tt.datcreat, tt.datmodif, tt.datannul, tt.devise, tt.drgl, tt.secondes, tt.typrgl, ;
      tt.exprj, tt.souclef, tt.ltrfisc, tt.clef, tt.codbanque, tt.tireur, tt.numchq, tt.bits1, tt.cleffac, ;
      tt.comment, tt.endoss, tt.quiendos, tt.souclemax, tt.cleprlvaut, ;
        (SELECT SUM(sr.montant) FROM paiement sr WHERE sr.clef= tt.clef AND sr.clef>0 AND ;
          sr.datmodif={} AND sr.datannul={}) AS mtotal ;
       FROM paiement tt ;
       WHERE (tt.anscol LIKE ?lcannee OR tt.ltrfisc LIKE ?lcannee) AND ;
       tt.clef BETWEEN ?lnclefmin AND ?lnclefmax AND ;
       tt.codfam LIKE ?lccodcou AND ;
       tt.drgl BETWEEN ?lddebut_periode AND ?ldfin_periode AND ;
       (tt.datremise = ?ldremise OR tt.datremise BETWEEN ?lddebut_remise AND ?ldfin_remise) AND ;
       tt.codbanque LIKE ?lccodbanque) rr ;
WHERE rr.mtotal BETWEEN ?lnmontant_min AND ?lnmontant_max OR ;
    rr.clef IN (SELECT in.clef FROM paiement in WHERE ;
                 (in.anscol LIKE ?lcannee OR in.ltrfisc LIKE ?lcannee) AND ;
                  in.clef BETWEEN ?lnclefmin AND ?lnclefmax AND ;
                  in.codfam LIKE ?lccodcou AND ;
                  in.drgl BETWEEN ?lddebut_periode AND ?ldfin_periode AND ;
                  (in.datremise = ?ldremise OR ;
                     in.datremise BETWEEN ?lddebut_remise AND ?ldfin_remise) AND ;
                  in.codbanque LIKE ?lccodbanque AND ;
                  in.montant BETWEEN ?lnmontant_min AND ?lnmontant_max) ;
ORDER BY drgl, secondes, codfactur
```