Comment augmenter les performances de requêtes DB2 répétitives avec la mémoïsation

Contents

1 Principes

La mémoïsation est une technique d'optimisation qui consiste à conserver les résultats d'une opération d'un appel à l'autre. L'objectif est donc d'accumuler les résultats déjà établis dans une structure de stockage (un cache) pour éviter de les recalculer au besoin.

Lorsqu'une requête sql mémoïsée est appelée plusieurs fois avec les mêmes paramètres, elle renvoie donc la valeur stockée dans le cache plutôt que de la demander à nouveau à DB2.

Une fonction peut être mémoïsée seulement si elle est déterministe, c'est-à-dire si sa valeur de retour ne dépend que de la valeur de ses arguments (sans effet de bord lié à d'autres paramètres cachés ou externes). Bien qu'une requête sql soit par essence non déterministe, les données pouvant être modifiées à tout moment par une tâche extérieure, on considérera cette propriété nécessaire et acquise (accès à des données référentielles pour la durée d'un traitement batch par exemple)

La structure de stockage d'une fonction mémoïsée repose le plus souvent sur une table de hachage remplie à la volée. Une fonction de hachage appliquée sur chaque clé calcule de manière rapide un indice permettant un accès direct aux données stockées en table

2 Mise en oeuvre en cobol

La fonction de mémoïsation est confiée à un module cobol imbriqué (nested program) auquel sont transmis en arguments la clé d'accès et les données attendues en retour (utilisation des host variables du DCLGEN de la table accédée dans le cadre de normes de développement standards).

La requête sql est "déportée" dans le nested program (sans aucune modification si l'écriture a été faite de façon "standard" par l'utilisation des variables du DCLGEN), et doit être remplacée dans le programme

principal par un `Call nested_program...`

L'algorithme est simple: le module va chercher le résultat de la requête dans la table du cache, s'il ne s'y trouve pas, il va le chercher dans la base de données et le stocke en table pour qu'il soit disponible plus rapidement pour les prochaînes requêtes.

La table cobol est allouée dynamiquement lors du premier appel au module, sa taille a été au préalable spécifiée par le développeur

Il n'est pas nécessaire de faire figurer la variable `SQLCODE` dans les arguments car elle bénéficie automatiquement d'une portée globale dans le programme principal

Le code de gestion de la table de hachage est totalement pris en charge par un copy cobol réutilisable. Le développeur doit simplement prendre en charge la description des colonnes accédées (les clés d'accès de la clause `WHERE` et les données rapatriées par la clause `INTO` de la requête sql)

Avantages procurés par l'utilisation d'une table de hachage remplie à la volée:

- Le coût de recherche demeure constant quel que que soit la taille de la table (en comparaison, le coût d'un `SEARCH ALL` cobol augmente avec log(n))
- Aucune contrainte n'est exigée sur l'ordre de stockage des clés (en comparaison, le `SEARCH ALL` cobol impose un tri des clés en table)
- La taille de la table ne constitue jamais une limite fonctionnelle au programme. Ainsi, si le domaine de valeur des clés venait à augmenter dans l'application au fil du temps, on observerait tout au plus une légère dégradations des performances (moins de données résolues dans le cache), mais en aucun cas un plantage du programme (là où une table cobol avec `SEARCH ALL`, préchargée en début de programme, induit généralement un plantage en cas de dépassement de capacité et l'obligation de modifier/recompiler le programme)

Avantages à isoler le code de mémoïsation dans un nested program:

- La notion de nested program a été introduite par IBM afin de procurer à cobol la même robustesse et souplesse que d'autres langages hautement structurés tels que C ou Pascal
- Un nested program, en bénéficiant d'une working-storage indépendante du programme principal, offre une sécurité et une maintenabilité accrue. Plusieurs nested program partageant des noms de variables identiques peuvent ainsi cohabiter au sain d'un même programme principal sans effet de bord
- Il devient possible de masquer du code techniques spécialisé et complexes en le séparant du code métier
- Certaines données peuvent être partagées (variable définie `GLOBALE`), c'est le cas notamment du `SQLCODE` (généré GLOBAL par le précompilateur)
- Malgré une "lourdeur apparente" pour le développeur dans la phase de codage, le compilateur sait transformer automatiquement tous les `CALL nested program` en `PERFORM` au moment de l'exécution.

3 Implantation dans un source existant: mode d'emploi

3.1 Substitution de la requête Sql dans le programme principal

Après avoir déterminer le nom du nested program, substituer la requête Sql (`Exec Sql ... End-exec`) par un `CALL <nested program>`. Les paramètres à lui transmettre sont le code action `"READ"` et la zone `DCLGEN` de la table :

```
.      Call "nested_program_name" Using By Content "READ"
                                       By Reference <DCLGENxxx>
```

(La clause `By Content "READ"` permet de faire l'économie d'une déclaration de constante en working)

Le `SQLCODE` est valorisé en retour du module de manière identique, et n'exige donc aucun changement dans le code source d'origine

3.2 Construction et paramétrage du nested program

Le nested program doit figurer à l'intérieur du programme principal, à la fin de celui-ci.

Il est construit à l'aide de 2 copys cobol dont un avec la clause `Replacing`

1. Coder une **Identification Division** en précisant le nom choisi pour le nested program

```
.      Identification Division.
       Program-id. <nested_program_name>.
```

2. La **Working-Storage** est construite à l'aide du copy `HASDB2W0` et complétée de 2 constantes et 1 variable:

```
.      Copy HASDB2W0.
       01 RESSOURCE-NAME      PIC X(8) Value "xxxxxxxx".
       01 HASH-TABLE-SIZE     PIC S9(8) binary Value nnnnnnnn.
       01 HASH-KEYS.
         05 '<col-clé-1  Pic ... >'
         .....
         05 '<col-clé-N  Pic ... >'
```

   xxxxxxxx = nom arbitraire pour la ressource gérée (sert uniquement à certains `DISPLAY` effectués en interne par le nested program)

   Paramètre important : nnnnnnnn = taille de la la table de hachage (En règle générale, si la table DB2 n'est pas trop volumineuse, il est préconisé de prendre la valeur du `count(*)` fourni par DB2)

   `HASH-KEYS` : cette zone groupe doit décrire chacune des colonnes clé présentes dans la clause `WHERE` de la requête sql à l'identique de celles présentes dans le DCLGEN de la table

3. Coder une **Linkage Section** suivant ce modèle :

```
.      Linkage Section.
      *================*
       01 HASH-FUNC PIC X(4).
      * Le copy du DCLGEN de la table :
       Copy DCLGENxxx Replacing '<nom_table>' By DUMMYxx.
      * La structure de la table de hachage :
       01 HASH-TABLE
         02 HASH-LINE Occurs 1 Depending On HASH-TABLE-SIZE
           03 HASH-TABLE-KEYS
      *      description des zones clé:
             05 '<col-clé-1  Pic ...>'
                 .....
             05 '<col-clé-N  Pic ...>'
```

```
           03 HASH-TABLE-DATA
      *      description des zones data:
             05 '<col-data-1 Pic ...>'
                .....
             05 '<col-data-N Pic ...>'
```

Si le `DCLGEN` est déjà utilisé dans le programme principal (généralement le cas) et s'il contient un `DECLARE table`, il est impératif de changer le nom de la table(`Replacing '<nom_table>' By DUMMYxx.`), le précompilateur n'acceptant pas la présence de 2 DECLARE TABLE identiques dans un même source

`HASH-TABLE-KEYS` : cette zone groupe doit décrire chacune des colonnes clé présentes dans la clause `WHERE` de la requête sql à l'identique de celles du DCLGEN de la table

`HASH-TABLE-DATA` : cette zone groupe doit décrire chacune des colonnes de donnée présentes dans la clause `INTO` de la requête sql à l'identique de celles du DCLGEN de la table

4. Construction de la **Procedure Division** à l'aide du copy `HASDB2P0` avec la clause `Replacing` + la requête SQL d'origine suivant ce modèle :

```
.      * Procédure Division:
        Copy HASDB2P0 Replacing
         ==:DCLGEN:==          By ==<DCLGENxxx>==

         ==:LOADKEYS:==        By ==
            Move '<col-clé-1>' Of '<DCLGENxxx>'
              To '<col-clé-1>' Of HASH-KEYS==
            ...
            Move '<col-clé-N>' Of '<DCLGENxxx>'
              To '<col-clé-N>' Of HASH-KEYS==

         ==:DATATOTABLE:==     By ==
            Move '<col-data-1>' Of '<DCLGENxxx>'
              To '<col-data-1>' Of HASH-TABLE-DATA(HASH)
            ...
            Move '<col-data-n>' Of '<DCLGENxxx>'
              To '<col-data-n>' Of HASH-TABLE-DATA(HASH)==

         ==:DATAFROMTABLE:==   By ==
            Move '<col-data-1>' Of HASH-TABLE-DATA(HASH)
              To '<col-data-1>' Of '<DCLGENxxx>'
            ...
            Move '<col-data-N>' Of HASH-TABLE-DATA(HASH)
              To '<col-data-N>' Of '<DCLGENxxx>'==.

       * requête SQL en provenance du programme principal:

          Exec sql
            SELECT  '<col_data_1>'
                   , ...
                   ,'<col_data_N>'
            INTO  '<:DCLGENxxx.col-data-1>'
                 , ...
                 ,'<:DCLGENxxx.col-data-N>'
            FROM  '<table>'
            WHERE '<col_clé_1>' = '<:DCLGENxxx.col-clé-1>'
              AND ...
              AND '<col_clé_N>' = '<:DCLGENxxx.col-clé-N>'
         End-exec
               .
        End Program <nested_program_name>.
        End Program <program_principal>.
```

Les 4 pseudo-variables permettent d'indiquer "le code d'alimentation" pour la clé et les données à stocker en table

`:DCLGEN:` sera substitué par le nom de la zone groupe `DCLGEN` de la table

`:LOADKEYS:` sera substitué par le code de chargement (`MOVEs`) de la clé du `DCLGEN` vers la variable interne `HASH-KEYS`

`:DATATOTABLE:` sera substitué par le code de chargement (`MOVEs`) des données du `DCLGEN` vers la table `HASH-TABLE-DATA`

`:DATAFROMTABLE:` sera substitué par le code de restitution (`MOVEs`) des données de la table `HASH-TABLE-DATA` vers le `DCLGEN`

4 Tuning et évaluation des performances

Les performances dépendent essentiellement d'un choix correct pour la taille de la table de hachage. Il n'est quelquefois pas possible d'évaluer à priori le nombre de requêtes répétitives émises par le programme principal. Pour aider à évaluer ce taux lors de l'exécution, le nested module dispose d'une fonction statistique affichant certains éléments utiles:

Le module doit être rappelé en fin de programme principal avec le code fonction "STAT":

```
.         Call "modulexx" Using By content   "STAT"
                              By reference DCLxxx
```

pour produire l'affichage de ce genre d'infos:

```
.      * Ressource name        :HASH0002
       * Hash table lines      :00000811
       * Taux de remplissage   :97%
       * Hash table size       :000089210 (bytes)
       * Hash factor limite    :00000027
       * Key len               :00000008
       * Module Call           :00002407
       * Exec SQL Call         :00000833
       * clés hashcodées       :00000787
       * clés rejetées         :00000046
       * Sqlcode rejetés       :00000000
       * Hash zero value       :00000000
       * collisions (hash-loop 00000000) :00000916
       * collisions (hash-loop 00000001) :00000637
       * collisions (hash-loop 00000002) :00000487
```

- Le rapport entre la valeur de `Module Call` et `Exec SQL Call` donne une excellente idée du gain procuré par le cache (nombre de requêtes "redondantes" effectuées par le programme principal prises en charge avec succés par le cache)
- Le taux de remplissage : idéalement il devrait être d'environ 80%, lorsqu'il est supérieur à 90%, la taille de la table devient insuffisante, s'il est trop faible, la taille de la table a été surdimensionnée (consommation inutile de mémoire)
- le nb de collisions et leur niveau `* collisions(hash-loop 000000nn) : nnnnnn` fournit une indication sur capacité pour la fonction de hash à trouver facilement un emplacement libre en table pour le stockage des clés. Un nombre trop élevé de collisions est souvent lié à une taille insuffisante de la table.

5 Quelques remarques

Note

- Il est important de vérifier la présence d'éventuelles directives de précompilation `SQL WHENEVER ...` en amont de l'appel au nested program, car elles n'auront plus l'effet escompté au retour du module. Il est impératif dans ce cas d'ajouter manuellement les tests afin d'obtenir un fonctionnement à l'identique (coder les équivalences de `NOT FOUND, SQLERROR ou SQLWARNING`)

Note

- Si les 2 pseudo-variables `:DATATOTABLE:` et `:DATAFROMTABLE:` ne sont pas valorisées (substituées par SPACES), seul le `SQLCODE` sera géré en table. Dans ce cas, le module permet de simuler un test d'existence (seul un `SQLCODE` sera renvoyé)

Note

- Une gestion particulière est opérée en cas de `SQLCODE -904` (indisponibilité dans DB2). Dans ce cas, les résultats ne sont pas mis en cache (et seront donc redemandés à DB2 la prochaine fois)

Note

- Toutes les clés présentées en entrée du module sont stockées dans le cache (y-compris celles retournant un `SQLCODE +100`), la taille du cache peut donc nécessiter une valeur plus élevée que celle du nombre de lignes stockées dans DB2

6 Exemple concret

Programme principal d'origine:

```
.       Identification Division.
        Program-id. LWEMIEXM.
           ...

           Exec sql
              SELECT  CODE_ENTITE_CRR,
                      CODE_APPLI_CRR
              INTO :DCLVLWBRCON.CODE-ENTITE-CRR,
                   :DCLVLWBRCON.CODE-APPLI-CRR
              FROM   VLWBRCONAPSI
              WHERE  BAAPID = :DCLVLWBRCON.BAAPID
```

```
            WITH UR
         End-exec

         EVALUATE SQLCODE
         WHEN 0 ...
         ...
```

Après modifications :

```
.      Identification Division.
       Program-id. LWEMIEXM.
          ...

          Call "SLWBRCON" Using By content   "READ"
                                By reference DCLVLWBRCON

          EVALUATE SQLCODE
          WHEN 0 ...
          ...

       Identification Division.
       Program-id. SLWBRCON.
      *Working-Storage:
       Copy HASDB2W0.
       01 RESSOURCE-NAME      PIC  X(8) Value "TLWBRCON".
       01 HASH-TABLE-SIZE     PIC S9(8) binary Value 4001.
       01 HASH-KEYS.
          05 TPS-DT-ARRT          PIC X(10).
          05 BAAPID               PIC X(12).

       Linkage Section.
      *================*
       01 HASH-FUNC           PIC X(4).
       copy LWBRCON Replacing VLWBRCONAPSI By DUMMY01.

       01 HASH-TABLE.
         02 HASH-LINE Occurs 1 Depending On HASH-TABLE-SIZE.
           03 HASH-TABLE-KEYS.
             05 TPS-DT-ARRT          PIC X(10).
             05 BAAPID               PIC X(12).
           03 HASH-TABLE-DATA.
             05 CODE-ENTITE-CRR      PIC X(5).
             05 CODE-APPLI-CRR       PIC X(12).

      *Procedure Division:
           Copy HASDB2P0 Replacing
            ==:DCLGEN:==          By ==DCLVLWBRCON==

            ==:LOADKEYS:==        By ==
               Move TPS-DT-ARRT Of DCLVLWBRCON
                 To TPS-DT-ARRT Of HASH-KEYS
               Move BAAPID      Of DCLVLWBRCON
                 To BAAPID      Of HASH-KEYS==

            ==:DATATOTABLE:==     By ==
               Move CODE-ENTITE-CRR Of DCLVLWBRCON
```

```
              To CODE-ENTITE-CRR Of HASH-TABLE-DATA(HASH)
            Move CODE-APPLI-CRR  Of DCLVLWBRCON
              To CODE-APPLI-CRR  Of HASH-TABLE-DATA(HASH)==

          ==:DATAFROMTABLE:==   By ==
            Move CODE-ENTITE-CRR Of HASH-TABLE-DATA(HASH)
              To CODE-ENTITE-CRR Of DCLVLWBRCON
            Move CODE-APPLI-CRR  Of HASH-TABLE-DATA(HASH)
              To CODE-APPLI-CRR  Of DCLVLWBRCON==.

         Exec sql
            SELECT CODE_ENTITE_CRR,
                   CODE_APPLI_CRR
            INTO :DCLVLWBRCON.CODE-ENTITE-CRR,
                 :DCLVLWBRCON.CODE-APPLI-CRR
            FROM VLWBRCONAPSI
            WHERE TPS_DT_ARRT = :DCLVLWBRCON.TPS-DT-ARRT
             AND  BAAPID      = :DCLVLWBRCON.BAAPID
         End-exec
               .
       End Program SLWBRCON.
       End Program LWEMIEXM.
```

.