Le MERGE sous SQL DB2

- Les articles -

Date de mise en ligne : samedi 14 juin 2014

Description :

Passé un peu inaperçu au milieu de toutes les nouveautés de la V7R1 de l'IBMi, l'ordre SQL MERGE est trop pratique pour que l'on puisse se permettre de passer à côté. Tour d'horizon de ses possibilités.

Environnement iSeries

Ces derniers mois, j'ai dépanné plusieurs confrères qui avaient tous en commun de devoir faire des mises à jour de tables SQL, dans le cadre de reprise ou de correction de données. A chaque fois le problème posé était le suivant : j'ai une table cible A dans laquelle je dois mettre à jour certaines lignes, à partir de données renvoyées par une requête source B. Enfin, quand je dis que je les ai dépannés... disons plutôt que je leur ai montré comment se servir de ce dont nous allons parler dans cet article, et je crois pouvoir affirmer que je les ai tous convertis... aux bienfaits du MERGE !

Traditionnellement, l'approche pour traiter ce genre de problème consiste à écrire un programme RPG ou Cobol, contenant une boucle de lecture sur la table cible. A l'intérieur de cette boucle, à chaque itération, on effectue une ou plusieurs lectures pour aller chercher les données dans une ou plusieurs tables sources, et on met à jour l'enregistrement dans la table cible avant de passer à l'enregistrement suivant. En RPG notamment, on utilise généralement les ordres de lecture natifs du langage, pour écrire la boucle de lecture et les opérations de mise à jour, même si certains développeurs - plus courageux - le font en utilisant un curseur SQL (je dis plus courageux car l'écriture via un curseur SQL est souvent plus verbeuse). A noter également que l'utilisation des ordres de lecture natifs (notamment les CHAIN) nécessitent la présence d'indexs adéquats pour l'accès aux données, ce qui, dans certains cas, peut compliquer l'opération de mise à jour (surtout quand il s'agit d'un "one shot" et qu'on est pressé).

Avec l'ordre DB2 SQL MERGE, plus besoin d'écrire un programme RPG ou Cobol, donc plus besoin de compilation et de livraison d'objet. Plus besoin non plus de créer des indexs adéquats s'ils font défaut. On va voir dans les exemples qui suivent que des opérations de mise à jour complexes peuvent être réalisées très rapidement, et en une seule requête SQL.

- Exemple 1 : mise à jour dans la table cible A de 3 colonnes (codea, coden et datmaj) dont 2 (les colonnes codea et coden) sont alimentées par le result set renvoyé par la requête définie dans la clause USING (avec 2 colonnes, macle1 et macle2, en conditions de jointure entre tables cible et source) :

```
MERGE INTO qtemp.testmerge2 a
USING (SELECT macle , codea , coden FROM qtemp.testmerge ) b
ON a.macle1 = b.macle1 and a.macle2 = b.macle2
WHEN MATCHED THEN
UPDATE SET
a.codea = b.codea ,
a.coden = a.coden + b.coden ,
a.datmaj = curdate()
;
```

- Exemple 2 : idem requête précédente, mais avec en supplément une insertion de ligne dans la table cible, si la condition de jointure entre table cible et requête source ne "matche" pas :

```
MERGE INTO qtemp.testmerge2 a
 USING (SELECT macle , codea , coden FROM qtemp.testmerge ) b
 ON a.macle1 = b.macle1 and a.macle2 = b.macle2
 WHEN MATCHED THEN
 UPDATE SET
 a.codea = b.codea ,
 a.coden = a.coden + b.coden ,
 a.datmaj = curdate()
 WHEN NOT MATCHED THEN
 INSERT ( a.macle , a.codea , a.coden, a.datmaj )
```

```
 VALUES( b.macle , b.codea , b.coden, curdate() )
 ;
```

Exemple 3 : utilisation de conditions complémentaires sur les clauses WHEN MATCHED, avec un DELETE sur la table cible quand la jointure principale "matche" et que la colonne a.codea = 'A1' :

```
MERGE INTO qtemp.testmerge2 a
 USING (SELECT macle , codea , coden FROM qtemp.testmerge ) b
 ON a.macle1 = b.macle1 and a.macle2 = b.macle2
 WHEN MATCHED and a.codea = 'A1'  THEN
 DELETE
 WHEN MATCHED and a.codea <> 'A1'  THEN
 UPDATE SET
 a.codea = b.codea ,
 a.coden = a.coden + b.coden ,
 a.datmaj = curdate()
 WHEN NOT MATCHED THEN
 INSERT ( a.macle , a.codea , a.coden, a.datmaj )
 VALUES( b.macle , b.codea , b.coden, curdate() )
 ;
```

On voit donc que l'on peut définir plusieurs blocs "WHEN MATCHED" avec des conditions différentes.

Exemple 4 : variante du cas précédent, avec cette fois un bloc WHEN MATCHED dans lequel des colonnes se voient fixer des valeurs "en dur" (quand la colonne a.codea = 'A2') :

```
MERGE INTO qtemp.testmerge2 a
 USING (SELECT macle , codea , coden FROM qtemp.testmerge ) b
 ON a.macle1 = b.macle1 and a.macle2 = b.macle2
 WHEN MATCHED and a.codea = 'A1'  THEN
 DELETE
 WHEN MATCHED and a.codea = 'A2'  THEN
 UPDATE SET
 a.codea = 'X2' ,
 a.coden = 9999 ,
 a.datmaj = curdate()
 WHEN MATCHED and a.codea <> 'A1' and a.codea <> 'A2'  THEN
 UPDATE SET
 a.codea = b.codea ,
 a.coden = a.coden + b.coden ,
 a.datmaj = curdate()
 WHEN NOT MATCHED THEN
 INSERT ( a.macle , a.codea , a.coden, a.datmaj )
 VALUES( b.macle , b.codea , b.coden, curdate() )
 ;
```

Maintenant quelques précisions.

Dans les exemples ci-dessus, j'ai défini une requête très basique à l'intérieur de la clauce USING, mais il est bien évident que vous pouvez définir dans cette clause des requêtes SQL beaucoup plus complexes, avec des jointures dans tous les sens, vous n'avez aucune contrainte à ce stade.

J'écrivais plus haut que la mise à jour en RPG via les ordres de lectures natifs nécessitaient la présence d'indexs adéquats. Avec le MERGE SQL, c'est le moteur SQL qui gère les accès aux données, donc si les indexs adéquats ne sont pas tous en place, le moteur DB2 s'en débrouillera. Si l'opération de mise à jour est urgente, je connais pas mal de développeurs qui seront contents de pouvoir se décharger sur le SGBD de la problématique des accès (et moi le premier).

Autre cas de figure que je n'ai trouvé référencé dans aucune documentation de DB2 for i, mais que j'ai découvert en expérimentant : vous avez des variables d'un programme RPG, ou d'un script PHP, ou d'une procédure stockée DB2, et vous voulez les utiliser pour effectuer des mises à jour dans une table sans passer par une table « source ». Vous pouvez dans ce cas utiliser une table pivot comme table source (comme SYSIBM.SYSDUMMY1) et écrire ceci (dans le cas du RPG, les variables programmes sont reconnaissables au fait qu'elles sont préfixées par « : ») :

```
MERGE INTO qtemp.testmerge A
 USING (SELECT * FROM SYSIBM.SYSDUMMY1) B
 ON A.macle = :VARPGM1
 WHEN MATCHED THEN
 UPDATE SET
 a.codea = :VARPGM1 ,
 a.coden = :VARPGM2 + 1
 WHEN NOT MATCHED THEN
 INSERT ( a.macle , a.codea , a.coden )
 VALUES(:VARPGM1, :VARPGM2 , 1 )
 ;
```

Si vous développez en SQL dynamique, vous êtes tout à fait en droit d'écrire ceci :

```
MERGE INTO qtemp.testmerge A
 USING (SELECT * FROM SYSIBM.SYSDUMMY1) B
 ON A.macle1 = ? AND A.macle2 = ?
 WHEN MATCHED THEN
 UPDATE SET
 a.codea = ? ,
 a.coden = ?
 WHEN NOT MATCHED THEN
 INSERT ( a.macle , a.codea , a.coden )
 VALUES( ? , ? , ? )
 ;
```

Ca fonctionne très bien, je peux en témoigner pour l'avoir utilisé aussi bien en PHP que dans des procédures stockées DB2 et des programmes RPG. Il faut simplement être vigilant à bien respecter dans la phase de l'EXECUTE l'ordre des paramètres correspondant à l'ordre des points d'interrogation tels qu'ils sont placés dans la requête. Si vous souhaitez utiliser cette technique dans une boucle pour mettre à jour une série de lignes de manière unitaire, je recommande de faire le "PREPARE" de la requête SQL en amont de la boucle de mise à jour. Cela permet d'obtenir des performances optimales.

Juste après avoir bouclé cet article, j'ai découvert dans la doc en ligne de DB2 pour Z/OS une variante de mon dernier exemple. Je viens de la tester sur DB2 for i, et comme elle fonctionne très bien, je vous la livre ci-dessous :

```
hv1 = "MERGE INTO employee AS t
 USING TABLE(VALUES(
 CAST (? AS CHAR(6)),
 CAST (? AS VARCHAR(12)),
```

```
CAST (? AS CHAR(1)),
CAST (? AS VARCHAR(15)),
CAST (? AS SMALLINT),
CAST (? AS INTEGER))
) s (empno, firstnme, midinit, lastname, edlevel, salary)
ON t.empno = s.empno
WHEN MATCHED THEN
UPDATE SET
salary = s.salary
WHEN NOT MATCHED THEN
INSERT
(empno, firstnme, midinit, lastname, edlevel, salary)
VALUES (s.empno, s.firstnme, s.midinit, s.lastname, s.edlevel, s.salary)";
EXEC SQL PREPARE s1 FROM :hv1;
EXEC SQL EXECUTE s1 USING '000420', 'SERGE', 'K', 'FIELDING', 18, 39580
;
```

Je trouve très astucieuse l'idée consistant à créer une table virtuelle à l'intérieur de la clause USING, table virtuelle qui accueille les données provenant des variables du programme. Cela permet de simplifier du même coup l'écriture du reste de la requête (on risque moins de se tromper dans l'ordre des points d'interrogation, qui ne sont plus disséminés dans l'ensemble de la requête). Avec ma méthode, on était obligé de "démultiplier" les points d'interrogation, en en plaçant à l'intérieur de chaque bloc WHEN MATCHED ou WHEN NOT MATCHED. La solution proposée dans la documentation de DB2 pour Z/OS est donc bien meilleure, et c'est tant mieux :)

En conclusion :

IBM nous a fait un formidable cadeau en incorporant dans la V7R1 l'ordre SQL MERGE. Si vous avez la chance de développer sur des serveurs qui sont au minimum en V7R1, je vous encourage vivement à l'expérimenter, et à l'intégrer dans vos développements.