Optimisation des LDD

1. Introduction

A partir de sa version 11g, Oracle a introduit une technique très performante qui permet l'optimisation des opérations d'ajout de colonnes sur une table. Lorsque vous ajoutez à une table une colonne **not nulle** et qu'en même temps vous attachez une valeur **par défaut** à cette colonne, alors une nouvelle technique d'optimisation LDD (**L**angage de **D**éfinition des **D**onnées) a lieu permettant à cette opération d'être instantanée. Comment cela est-il possible lorsque la table ainsi modifiée possède des millions d'enregistrements qui doivent voir leur nouvelle colonne ajoutée mise à jour avec la valeur par défaut? Et est-ce que cette nouvelle technique d'optimisation des opérations LDD a été implémentée sans aucun effet secondaire que nous devrions connaitre avant son utilisation ? C'est ce que je vais vous expliquer dans cet article.

2. Le Concept

Considérons la table suivante contenant 3 millions d'enregistrements

```
SQL> create table t1

     as select
        rownum n1
      , trunc ((rownum-1)/3) n2
      , trunc(dbms_random.value(rownum, rownum*10)) n3
      , dbms_random.string('U', 10) c1
    from dual
    connect by level <= 3e6;

SQL> desc t1
            Name              Null? Type
            ----------------------- ---------
    1       N1                      NUMBER
    2       N2                      NUMBER
    3       N3                      NUMBER
    4       C1                      VARCHAR2(4000 CHAR)
```

Table à laquelle je vais ajouter une colonne supplémentaire non nulle et ayant une valeur par défaut. Quelque chose comme ceci:

```
SQL> alter table t1 add C_DDL number default 42 not null;
```

Où j'ai mis en gras les deux plus importants mots à savoir **default** et **not null** parce qu'ils représentent les clauses qui gèrent cette nouveauté.

Pour mieux apprécier la différence dans le temps d'exécution de l'instruction '*alter table*' ci-dessus, je vais l'exécuter dans deux différentes versions de la base de données Oracle, 10.2.0.4.0 et 11.2.0.3.0 respectivement:

```
10.2.0.4.0 > alter table t1 add C_DDL number default 42 not null;
```

```
Table altered.
Elapsed: 00:00:48.53

11.2.0.3.0> alter table t1 add C_DDL number default 42 not null;

Table altered.
Elapsed: 00:00:00.04
```

Observez la différence dans le temps d'exécution. La colonne `C_DDL` a été ajoutée instantanément dans la version 11gR2 alors que son ajout dans la version 10gR2 a nécessité 49 secondes. Quel est donc ce nouveau mécanisme qui permet cet extrêmement rapide temps d'exécution lors de l'ajout d'une colonne non nulle ayant une valeur par défaut à une table existante?

Comment 3 millions d'enregistrements peuvent être mis à jour en 4 millisecondes?

Vérifions visuellement si cet update a été réellement fait (à partir de maintenant lorsque la version d'Oracle n'est pas précisée il s'agira alors implicitement de la version 11.0.2.3)

```
SQL> select count(1) from t1;

  COUNT(1)
----------
   3000000

SQL> select count(1) from t1 where c_ddl = 42;

  COUNT(1)
----------
   3000000
```

Bien qu'Oracle ait modifié la table t1 instantanément, la requête montre que la totalité des colonnes `C_DDL` a été mise à jour avec sa valeur par défaut 42. Comment cela est-il possible? Est-ce que le plan d'exécution peut nous aider ici?

```
SQL> select * from table(dbms_xplan.display_cursor);

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |       |       |  3016 (100)|          |
|   1 |  SORT AGGREGATE    |      |     1 |     3 |            |          |
|*  2 |   TABLE ACCESS FULL| T1   |  2999K|  8788K|  3016   (5)| 00:00:10 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter(NVL("C_DDL",42)=42)
```

Notez bien encore ici comment la partie prédicat du plan d'exécution précédent peut révéler des informations capitales pour la compréhension de ce qui se passe derrière la scène. Je n'ai pas utilisé la fonction **`NVL`** dans ma requête, mais elle apparait quand même dans la partie prédicat indiquant qu'Oracle considère encore que la colonne `C_DDL` est susceptible de contenir des valeurs nulles (ce qui signifie, qu'en réalité, cette colonne n'a pas été mise à jour, et c'est la raison pour laquelle, Oracle est en train de la remplacer par sa valeur par défaut 42)

Nous avons la version précédente d'Oracle à notre disposition pour comprendre et pour localiser les différences :

```
10.2.0.4.0 > select count(1) from t1 where c_ddl = 42;

  COUNT(1)
----------
   3000000

10.2.0.4.0> select * from table(dbms_xplan.display_cursor);

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |       |       |  4001 (100)|          |
|   1 |  SORT AGGREGATE    |      |     1 |     3 |            |          |
|*  2 |   TABLE ACCESS FULL| T1   |  3000K|  8789K|  4001   (8)| 00:00:09 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("C_DDL"=42)
```

L'absence de la fonction **NVL** dans la partie prédicat couplée au temps qui a été nécessaire pour l'ajout de la colonne C_DDL dans 10gR2 (00:00:**48.53**) expliquent le fonctionnement du concept, introduit en 11gR1, d'optimisation des ajouts de colonnes non nulles ayant une valeur par défaut à une table existante.

A partir de la version 11gR1 d'Oracle, lorsqu'on ajoute une colonne **non nulle** ayant une valeur par **défaut**, Oracle ne vas pas mettre à jour tous les enregistrements existants avec cette valeur par défaut. Il va, par contre, stocker un méta-data pour cette nouvelle colonne (contrainte non nulle et valeur par défaut 42) et va permettre ainsi au LDD d'être accomplie instantanément, et ce, quelle qu'ait été la taille de la table modifiée. Bien sûr, ceci est possible moyennant l'utilisation de la fonction **NVL** lors de la lecture de cette nouvelle colonne à partir d'un bloc de la table.

Après avoir expliqué ce magnifique concept d'optimisation LDD, je vais, dans le paragraphe suivant, investiguer un peu plus comment cette nouveauté est gérée par Oracle pour assurer une rapidité de l'opération LDD et pour garantir un résultat correct et performant lors de la sélection des données. Nous allons particulièrement voir que la sélection de la colonne ajoutée à partir d'un bloc de table diffère de celle faite à partir d'un bloc feuille (leaf block) d'un index.

3. Effet du mécanisme LDD

3.1. Sur la table modifiée

Nous avons vu plus haut que nous gagnons en performance lors de l'ajout d'une colonne non nulle ayant une valeur par défaut. Par contre, nous avons vu aussi que ceci est possible parce qu'Oracle utilise la fonction NVL pour l'appliquer à la nouvelle colonne C_DDL afin de lui attacher sa valeur par défaut. Ceci est possible grâce au méta-data de la colonne C_DDL qui a été stocké dans le dictionnaire des données. Est-ce que l'utilisation de cette fonction NVL génère un effet secondaire ?

Premièrement nous avons vu précédemment que ceci n'a aucune influence sur les estimations faites par le CBO qui continue dans ce cas à faire des estimations très précises du nombre de lignes à générer comme montré ci-dessous :

```
SQL> select /*+ gather_plan_statistics */ count(1) from t1 where C_DDL = 42;

  COUNT(1)
----------
   3000000

SQL> select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |        |      1 |00:00:00.37 |
|   1 |  SORT AGGREGATE    |      |      1 |      1 |      1 |00:00:00.37 |
|*  2 |   TABLE ACCESS FULL| T1   |      1 |   2999K|   3000K|00:00:00.44 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter(NVL("C_DDL",42)=42)
```

Par contre, en scannant les blocs de la table `T1` on remarque l'existence d'un temps d'exécution supplémentaire (`44ms`) par rapport à celui de la même opération dans la version d'Oracle précédente (`5ms`). Ceci est probablement dû à l'application du nouveau filtre qui utilise la fonction `NVL` :

```
10.2.0.4.0> select /*+ gather_plan_statistics */ count(1) from t1 where C_DDL = 42

  COUNT(1)
----------
   3000000

10.2.0.4.0> select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS
LAST'));

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Starts | E-Rows | A-Rows |   A-Time   |
-------------------------------------------------------------------------------
|   1 |  SORT AGGREGATE    |      |      1 |      1 |      1 |00:00:01.06 |
|*  2 |   TABLE ACCESS FULL| T1   |      1 |   3000K|   3000K|00:00:00.05 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("C_DDL"=42)
```

3.2. Sur la colonne C_DDL indexée

Lorsqu'une fonction est appliquée sur une colonne figurant dans une clause '*where*' de la partie prédicat, elle empêche toute utilisation d'un index qui peut éventuellement exister sur cette colonne. Dans le cas particulier qui nous concerne ici, est-ce que l'utilisation de la fonction `NVL` appliquée à la

colonne va empêcher un index d'être utilisé par le CBO si cette colonne est indexée? C'est ce que nous allons voir ci-après.

Considérons l'index suivant:

```
SQL> create index i1_c_ddl on t1(c_ddl);

Index created.

Elapsed: 00:00:02.14
```

Et ré-exécutons la même requête encore une fois :

```
SQL> select /*+ gather_plan_statistics */ count(1) from t1 where C_DDL = 42;

  COUNT(1)
----------
   3000000


SQL> select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));

-------------------------------------------------------------------------------------
| Id  | Operation             | Name     | Starts | E-Rows | A-Rows |   A-Time   |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |          |      1 |        |      1 |00:00:00.47 |
|   1 |  SORT AGGREGATE       |          |      1 |      1 |      1 |00:00:00.47 |
|*  2 |   INDEX FAST FULL SCAN| I1_C_DDL |      1 |   2999K|   3000K|00:00:00.75 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("C_DDL"=42)
```

Il y a une bonne nouvelle ici : l'index est utilisé. La fonction cachée `NVL` n'est pas appliquée sur la colonne `C_DDL` lorsque celle-ci provient de l'index. Ceci explique pourquoi l'index a bien été utilisé par le CBO.

Néanmoins, vous pouvez objecter en disant que ceci est un fonctionnement normal et attendu : les valeurs nulles d'une colonne ne sont pas indexées. C'est pourquoi nous allons maintenant créer un index composé de deux colonnes dont l'une est non nulle ; ainsi toutes les valeurs de la colonne `C_DDL`, y compris les valeurs nulles, seront indexées. Quelque chose qui ressemble à ce qui suit :

```
SQL> drop index i1_c_ddl;

Index dropped.

SQL> alter table t1 modify n1 not null;

Table altered.

SQL> create index i2_n1_c_ddl on t1(n1,c_ddl);

Index created.
```

```
SQL> select /*+ gather_plan_statistics */ count(1) from t1 where n1= 100 and C_DDL
= 42;

  COUNT(1)
----------
         1
-------------------------------------------------------------------------------------
| Id  | Operation         | Name        | Starts | E-Rows | A-Rows |   A-Time   |
-------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |             |      1 |        |      1 |00:00:00.01 |
|   1 |  SORT AGGREGATE   |             |      1 |      1 |      1 |00:00:00.01 |
|*  2 |   INDEX RANGE SCAN| I2_N1_C_DDL |      1 |      1 |      1 |00:00:00.01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("N1"=100 AND "C_DDL"=42)
```

Même lorsque la nouvelle colonne ajoutée `C_DDL` est protégée contre les valeurs nulles grâce à sa présence dans un index composé, on ne trouve aucune trace de l'utilisation implicite de la fonction `NVL` appliquée à la colonne `C_DDL`. Ceci démontre clairement que, à l'inverse des blocs de la table où aucun update de la colonne `C_DDL` n'a été fait, un index crée sur la même colonne va voir ses blocs feuilles (leaf blocks) immédiatement mis à jour par la valeur par défaut de la colonne `C_DDL`.

Avant de finir ce paragraphe je vais vous montrer un autre point intéressant à connaitre : nous avons vu plus haut qu'à chaque fois que le CBO décide de visiter un bloc de la table, il applique alors la fonction `NVL` à la colonne `C_DDL` pour être sûr de rapatrier des valeurs non nulles de cette colonne (parce qu'en effet celle-ci n'a pas été mise à jour). Mais nous avons vu que ce filtre est toujours appliqué lorsque la table est totalement scannée (`TABLE ACCESS FULL`). Est-ce que le CBO va également appliquer cette fonction lorsque la table t1 est accédée via un index (`TABLE ACCESS BY INDEX ROWID`)? Je vais donc modéliser un cas très simple pour observer la réaction du CBO dans cette situation particulière :

```
SQL> drop index i2_n1_c_ddl;

SQL> create index i2_n1_c_ddl on t1(n1);

SQL> select /*+ gather_plan_statistics */ count(1) from t1 where n1= 100 and C_DDL
= 42;

-------------------------------------------------------------------------------
| Id  | Operation                   | Name        | Starts | E-Rows | A-Rows |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT            |             |      1 |        |      1 |
|   1 |  SORT AGGREGATE             |             |      1 |      1 |      1 |
|*  2 |   TABLE ACCESS BY INDEX ROWID| T1          |      1 |      1 |      1 |
|*  3 |    INDEX RANGE SCAN         | I2_N1_C_DDL |      1 |      1 |      1 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter(NVL("C_DDL",42)=42)
   3 - access("N1"=100)
```

Observez comment la fonction `NVL` a été également appliquée sur la colonne même lorsque la table t1 est visitée via *index rowid*.

Nous sommes très confiants maintenant de dire qu'à chaque fois que le CBO visite un bloc à partir de la table, que cette visite soit faite via une lecture par bloc unique ou via un accès multi-bloc, il va appliquer la fonction `NVL` à la colonne ''LDD optimisée'' pour filtrer les données prises de ces blocs de table. Par contre, le CBO ne vas pas utiliser la fonction `NVL` sur la colonne `C_DDL` lorsque celle-ci est acquise à partir d'un bloc feuille d'un index.

4. Oracle 12c and DDL optimization for NULL columns

Avec l'arrivée de la version Oracle 12c c'est légitime de se demander si l'optimisation des instructions LDD est encore disponible ou pas. Une image valant mieux que milles mots, essayons aussi la même expérience dans cette nouvelle release :

```
12c > alter table t1 add C_DDL number default 42 not null;

Elapsed: 00:00:00.02
```

Presque instantanée. L'optimisation LDD a lieu ici aussi comme montré et prouvé encore une fois par le biais de l'utilisation de la fonction `NVL` dans la partie prédicat de la requête suivante :

```
12c> select count(1) from t1 where c_ddl=42;

  COUNT(1)
----------
   3000000

12c> select * from table(dbms_xplan.display_cursor);

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |       |       |  3802 (100)|          |
|   1 |  SORT AGGREGATE    |      |     1 |    13 |            |          |
|*  2 |   TABLE ACCESS FULL| T1   |  3538K|    43M|  3802   (1)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter(NVL("C_DDL",42)=42)

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)
```

Par contre, il y a, néanmoins, une petite extension de l'optimisation LDD en 12c par rapport à 11gR2. Dans la nouvelle version, cette technique a été étendue pour inclure les colonnes **nulles** ayant une valeur par **défaut**. Considérons la modification de la table suivante faite en 11gR2 et en 12c respectivement pour apprécier clairement la différence :

```
11.2.0.3.0> alter table t1 add C_DDL_2 number default 84;

Table altered.
```

```
Elapsed: 00:00:58.25

12c> alter table t1 add C_DDL_2 number default 84;

Elapsed: 00:00:00.02
```

Alors que l'ajout de la colonne `C_DDL_2` (qui peut être nulle) a nécessité 58 secondes en 11gR2, il a été accompli instantanément dans la version 12c.

Ceci représente une démonstration claire qu'en 12c, l'optimisation des opérations LDD a été étendue aux colonnes **nulles** ayant une valeur par défaut. En effet, lorsqu'on visite la table t1 pour avoir les valeurs distinctes de la nouvelle colonne ajoutée (`C_DDL_2`) on se rend compte que tous les enregistrements de la table ont vu leur métadonnées (valeur par défaut 84) mis à jour comme montré par le biais de la requête suivante :

```
12c> select c_ddl_2, count(1) from t1 group by c_ddl_2;

C_DDL_2 COUNT(1)
------- ----------
84      3000000


SQL> select count(1) from t1 where c_ddl_2=84;

  COUNT(1)
----------
   3000000

SQL> select * from table(dbms_xplan.display_cursor);

-------------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |       |       |  3803 (100)|          |
|   1 |  SORT AGGREGATE    |      |     1 |    13 |            |          |
|*  2 |   TABLE ACCESS FULL| T1   |  3538K|    43M|  3803   (1)| 00:00:01 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter(DECODE(TO_CHAR(SYS_OP_VECBIT("SYS_NC00006$",0)),NULL,NVL("
              C_DDL_2",84),'0',NVL("C_DDL_2",84),'1',"C_DDL_2")=84)

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)
```

Cependant, afin d'implémenter l'optimisation LDD pour les colonnes pouvant être nulles, une légère complexité a été introduite par rapport à l'optimisation LDD des colonnes non nulles dans la version précédente d'Oracle. Au lieu et place de la simple utilisation de la fonction `NVL`, est apparu un prédicat exotique et complexe utilisant la fonction d'Oracle `SYS_OP_VECBIT` non documentée et une colonne interne afin d'honorer la valeur par défaut puisque celle-ci n'a pas été physiquement mise à jour.

Contrairement à ce que vous pouvez immédiatement supposer, la colonne SYS_NC00006$ n'est pas virtuelle. Elle représente une colonne cachée générée intérieurement par Oracle comme montré ci-dessous :

```
12c> SELECT
           column_name
          ,virtual_column
          ,hidden_column
          ,user_generated
     FROM
           user_tab_cols
     WHERE table_name = 'T1'
     AND   column_name = 'SYS_NC00006$';

COLUMN_NAME          VIR HID USE
-------------------- --- --- ---
SYS_NC00006$         NO  YES NO
```

Bien que cette colonne soit cachée, ceci ne nous empêche pas de la sélectionner:

```
12c> select
        a.c_ddl_2
       ,a.SYS_NC00006$
     from t1 a
     where c_ddl_2 =84
     and rownum <=5;

C_DDL_2 SYS_NC00006$
------- ------------
     84
     84
     84
     84
     84
```

La colonne SYS_NC00006$ va rester nulle jusqu'à ce que la colonne C_DDL_2 reçoive une valeur qui n'est pas égale à la valeur par défaut 84. Considérons les inserts suivants :

```
12c> insert into t1 values (0,0,0,'xxxxx',110,130);

1 row created.

12c> insert into t1 values (1,1,1,'xxxxx',140,150);

1 row created.

12c> insert into t1 values (1,1,1,'xxxxx',200,null);

12c> select
         a.c_ddl_2
        ,a.SYS_NC00006$
     from t1 a
     where a.c_ddl_2 in (130,150);

C_DDL_2 SYS_NC00006$
------- ------------
    130 01
    150 01
```

```
SQL> select
        a.c_ddl_2
       ,a.SYS_NC00006$
     from t1 a
     where a.c_ddl_2 is null;

C_DDL_2 SYS_NC00006$
------- ------------
        01
```

Observez comment la valeur de la colonne cachée `SYS_NC00006$` n'est plus `NULL` lorsqu'on insert une valeur différente de la valeur par défaut 84 (y compris la valeur explicite `NULL`)

En mettant les différents pièces du puzzle ensemble, nous pouvons très facilement comprendre que cet exotique, mais néanmoins simple, prédicat, reproduit ci-dessous, est en train de faire exactement :

```
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter(DECODE(TO_CHAR(SYS_OP_VECBIT("SYS_NC00006$",0)),NULL,NVL("
              C_DDL_2",84),'0',NVL("C_DDL_2",84),'1',"C_DDL_2")=84)
```

Oracle est simplement en train de vérifier à travers, sa colonne système, ainsi que par le biais de l'utilisation de la fonction, s'il doit prendre en considération la valeur par défaut de la colonne ou bien sa vraie valeur introduite par un utilisateur final ou via une requête d'insertion explicite. Essayons d'imiter ce qu'Oracle est en train de faire avec les valeurs '`01`' and `NULL` ci-dessus de la colonne `SYS_NC00006$`

```
12c> SELECT
             a.c_ddl_2
            ,TO_CHAR(sys_op_vecbit(a.sys_nc00006$,0)) cbo_ddl
     FROM t1 a
     WHERE a.c_ddl_2 IN (130,150)
   UNION ALL
     SELECT
             a.c_ddl_2
            ,TO_CHAR(sys_op_vecbit(a.sys_nc00006$,0)) cbo_ddl
     FROM t1 a
     WHERE a.c_ddl_2 IS NULL
   UNION ALL
     SELECT
            a.c_ddl_2
           ,TO_CHAR(sys_op_vecbit(a.sys_nc00006$,0)) cbo_ddl
     FROM t1 a
     WHERE c_ddl_2 =84
     AND rownum   <=1
    order by c_ddl_2 nulls last
    ;

 C_DDL_2    CBO_DDL
---------- ---------
  84       {null}
  130       1
  150       1
 {null}     1
```

Il existe 4 valeurs distinctes de la colonne `C_DDL_2`, la valeur par défaut (84) et 3 autres valeurs insérées explicitement 130,150 et `NULL`. Lorsqu'on utilise un prédicat sur la colonne pour rapatrier une ligne à partir d'un bloc de la table, le CBO d'Oracle va décoder la valeur de la colonne `CBO_DDL` ci-dessus (basée sur `SYS_NC00006$`) pour vérifier sa valeur par rapport à la valeur du paramètre de la requête (que ce paramètre soit transmis en dur ou en variable de liaison i.e. literal or bind variable). Ainsi, Oracle peut imiter correctement toutes les valeurs de la colonne `C_DDL_2` y compris celles ayant la valeur par défaut (84) et qui n'ont pas été physiquement mise à jour pour refléter cette valeur par défaut.

4. Conclusion

Oracle 11gR1 a introduit une magnifique nouveauté qui a fait en sorte qu'on ne soit plus jamais inquiet sur la continuité de notre application lorsqu'on projette d'ajouter, en temps réel, une colonne non nulle possédant une valeur par défaut

Cette nouveauté, appelée 'DDL optimization', permet d'ajouter une colonne ayant une valeur par défaut à une table, non seulement instantanément, mais également sans avoir besoin de poser un verrou sur la table altérée. Oracle 12c a étendu cette technique pour y inclure les colonnes nulles avec une valeur par défaut. Et, cerise sur le gâteau, il semble que ceci soit sans effets notables sur la performance que l'on est sensée observer lors de la sélection de cette colonne ajoutée. En 12c l'apparition d'un prédicat exotique appliqué lorsque la colonne est lue à partir d'un bloc de la table s'est lui aussi avéré inoffensif et sans effet notable sur la performance d'un select sur les blocs de données de la table.

Mohamed Houri has a PhD in Fluid Mechanics (Scientific Computing) from the University of Aix-Marseille II, preceded by an engineer diploma in Aeronautics. He has been working around the Oracle database for more than 14 years for different Europeans customers as an independent Oracle Consultant specialized in Tuning and Trouble-shooting Oracle performance problems. Mohamed has also worked with the Naval Architect Society of Japan on the analysis of tsunamis and breaking waves using a powerful signal analysis called Wavelet Transform. He maintains an Oracle blog and is active in the Oracle Worldwide forum and in the French equivalent. He tweets about Oracle stuff at @MohamedHouri. Mohamed also is a member of OraWorld Team (www.oraworld-team.com).