DB2 Viper 2 compatibility features

Make your application ports easier

Serge Rielau

Porting an application with CONNECT BY, NVL, or other vendor specific SQL to DB2? Do not despair. DB2 Viper 2 for Linux, UNIX, and Windows understands. This article describes syntax and semantics added to DB2 to speak that other tongue.

Motivation

If you have ever worked with more than one relational data server, you know that SQL is not one language. In fact, SQL standard is an artificial construct not supported by any major player. Every vendor has their own idiosyncrasies, extensions, and shortcomings.

For you, as an application developer, that means you are left with a choice:

• Limit your SQL usage to the absolute minimum required (typically barely more than SELECT * FROM T) and do all meaningful work in the application. That sure defeats the purpose of SQL and bypasses the powerful optimization techniques for which you or your customers have already paid.
• Exploit the data server and its proprietary capabilities to the fullest and deal with the consequences. Among these consequences you will find that you have become hostage to the data server vendor. It is hard to quibble with the dealer on price when he knows you need your fix and you have to come to him. Your threats to leave will sound hollow at best.
• Shades of gray of the above. Typically you choose some sort of abstraction layer that allows you to customize data access at some level of complexity. This choice requires a great deal of restraint.

Unfortunately, falling into this second category happens quite easily and often remains undetected. Did you know, for example, that it is impossible to write a query with something as simple as an inline view in Oracle® and have it work against DB2, or the other way around?

This article introduces a grab bag full of tweaks and features in DB2 Viper 2 that greatly extend the scope of queries written against Oracle that, out of the box, run against DB2. None of these
features show up in marketing slides, but they just might be what makes the difference to you when porting to DB2.

**Set the mood**

Some of the features described in this article are bound to be controversial. This may not only be because they are non-standard, but also because they are either obsolete to begin with and therefore not allowed under coding guidelines, or the features may be in outright conflict with DB2’s behavior. In order to accommodate backward compatibility, and allow a level of control for the DBA over what is being used, these particular features have been placed under registry control. The DB2 registry variable is called `DB2_COMPATIBILITY_VECTOR` and it is set to a hex string. To turn on all features, perform the following set-up from a shell:

**Listing 1. Enable all compatibility features**

```sql
db2stop
db2set DB2_COMPATIBILITY_VECTOR=0F
db2start
```

Which specific bit corresponds to which individual feature are highlighted in this article as topics are introduced.

**Recursion**

The article "Port CONNECT BY to DB2" (developerWorks, Oct 2006) discussed how to map Oracle style recursion using the `CONNECT BY` clause to SQL standard recursive common table expressions using `WITH` and `UNION ALL`. It even proved that SQL standard recursion is at least as powerful as `CONNECT BY`. In fact, the standard notation is even more powerful given that it can effectively produce more rows than provided by the input data.

This power is fine, but when porting an application, technology is not the issue. It turns out that some applications rely heavily on recursion. In those cases, the effort of translating from `CONNECT BY` to SQL standard recursion alone can become prohibitively expensive. This is especially true if the application relies on the implied ordering produced by `CONNECT BY`.

For that reason, DB2 Viper 2 introduces built-in support for `CONNECT BY` recursion.

When looking at a given language syntax, both expressive power and simplicity are of importance. Unfortunately, these properties collide and recursion shows this clearly:

- The SQL standard recursion is extremely powerful using only a negligible language extension:
  1. Allow reference of the table expression within its own definition.
  2. Use `UNION ALL` to model the seed of a recursion and the recursive step.
- On the other hand, the `CONNECT BY` recursion is very user friendly for a common case: walking of hierarchies.

This section discusses not only how `CONNECT BY` works, but also attempts to describe when best to use which style of recursion.
The easiest example to explain, and perhaps the most popular usage scenario for CONNECT BY, is a reports-to chain in a company. Assume the following:

### Listing 2. Employee table for CONNECT BY

```sql
CREATE TABLE emp(empid INTEGER NOT NULL PRIMARY KEY,
    name VARCHAR(10),
    salary DECIMAL(9, 2),
    mgrid INTEGER REFERENCES emp(empid));

INSERT INTO emp VALUES
    (1, 'Jones', 30000, 10),
    (2, 'Hall', 35000, 10),
    (3, 'Kim', 40000, 10),
    (4, 'Lindsay', 38000, 10),
    (5, 'McKeough', 42000, 11),
    (6, 'Barnes', 41000, 11),
    (7, 'O''Neil', 36000, 12),
    (8, 'Smith', 34000, 12),
    (9, 'Shoeman', 33000, 12),
    (10, 'Monroe', 50000, 15),
    (11, 'Zander', 52000, 16),
    (12, 'Henry', 51000, 16),
    (13, 'Aaron', 54000, 15),
    (14, 'Scott', 53000, 16),
    (15, 'Mills', 70000, 17),
    (16, 'Goyal', 80000, 17),
    (17, 'Urbassek', 95000, NULL);
```

The following query returns all the employees working for Goyal as well as some additional information, such as the reports-to chain:

### Listing 3. CONNECT BY example

```sql
1 SELECT name,
2        LEVEL,
3        salary,
4        CONNECT_BY_ROOT name AS root,
5        SUBSTR(SYS_CONNECT_BY_PATH(name, ':'), 1, 25) AS chain
6   FROM emp
7   START WITH name = 'Goyal'
8   CONNECT BY PRIOR empid = mgrid
9   ORDER SIBLINGS BY salary;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>LEVEL</th>
<th>SALARY</th>
<th>ROOT</th>
<th>CHAIN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goyal</td>
<td>1</td>
<td>80000.00</td>
<td>Goyal</td>
<td>:Goyal</td>
</tr>
<tr>
<td>Henry</td>
<td>2</td>
<td>51000.00</td>
<td>Goyal:Henry</td>
<td>:Goyal:Henry:Henry</td>
</tr>
<tr>
<td>Shoeman</td>
<td>3</td>
<td>33000.00</td>
<td>Goyal:Henry:Shoeman</td>
<td>:Goyal:Henry:Shoeman:Shoeman</td>
</tr>
<tr>
<td>Smith</td>
<td>3</td>
<td>34000.00</td>
<td>Goyal:Henry:Smith</td>
<td>:Goyal:Henry:Smith:Smith</td>
</tr>
<tr>
<td>Zander</td>
<td>2</td>
<td>52000.00</td>
<td>Goyal:Zander</td>
<td>:Goyal:Zander:Zander</td>
</tr>
<tr>
<td>Barnes</td>
<td>3</td>
<td>41000.00</td>
<td>Goyal:Zander:Barnes</td>
<td>:Goyal:Zander:Barnes:Barnes</td>
</tr>
<tr>
<td>McKeough</td>
<td>3</td>
<td>42000.00</td>
<td>Goyal:Zander:McKeough</td>
<td>:Goyal:Zander:McKeough:McKeough</td>
</tr>
<tr>
<td>Scott</td>
<td>2</td>
<td>53000.00</td>
<td>Goyal:Scott</td>
<td>:Goyal:Scott:Scott</td>
</tr>
</tbody>
</table>

9 record(s) selected.

The individual pieces of the query have the following meaning:

- Lines 7 and 8 comprise the core of the recursion. The optional **START WITH** describes the WHERE clause to be used on the source table for the seed of the recursion. In this case, you
select only the row of employee Goyal. If START WITH is omitted, the entire source table is the seed of the recursion.

- **CONNECT BY** describes how, given the existing rows, the next set of rows is to be found. To distinguish values from the previous step with those from the current step, CONNECT BY recursion uses a unary operator **PRIOR**. PRIOR describes EMPID to be the employee ID of the previous recursive step, while MGRID originates from the current recursive step. As a unary operator, PRIOR has the highest precedence. However, using parentheses the operator can cover an entire expression.

- **LEVEL** in line 2 is a pseudo column that describes the current level of recursion. Pseudo columns are a concept foreign to the SQL standard. If the resolution of an identifier in DB2 fails completely, meaning there is neither a column nor a variable with that name in scope at all, then DB2 considers pseudo columns.

- **CONNECT_BY_ROOT** is another unary operator. It always returns the value of its argument as it was for the first recursive step. That is the values returned by the START WITH clause.

- **SYS_CONNECT_BY_PATH()** is a binary function. It prepends the second argument to the first, and then appends the result to the value it produced in the previous recursive step. Its arguments must be character types. Note that in DB2 the length of the result type is the greater of 1024 bytes and the length of the second argument.

Unless explicitly overridden, CONNECT BY recursion returns a result set in a partial order. That is, the rows produced by a recursive step always follow the row that produced them. Siblings at the same level of recursion with the same parent have no specific order.

- **ORDER SIBLINGS BY** in line 9 defines an order for siblings that further refines the partial order potentially into a total order.

Not shown in this example is that CONNECT BY recursion raises an error if a cycle occurs. A cycle happens when a row produces itself directly or indirectly. Using the optional CONNECT BY **NOCYCLE** clause, the recursion can ignore the duplicated row and therefore avoid the cycle and the error. DB2 for Viper 2 support 64 levels of recursion for CONNECT BY.

There is one caveat that you must be aware of when dealing with CONNECT BY. The semantics of the **WHERE** clause in conjunction with implicit joins.

- Any search condition that is used to join two tables in the FROM clause (such as S.pk = T.fk) is executed before the recursion is executed. Therefore, it is a filter reducing the rows that are being recursed over.
- Any local search condition (such as c1 > 5) is being executed on the result of the recursion.

Presumably this behavior is rooted in the history of the (+) join syntax.

Because of this interesting quirk, CONNECT BY has been placed under registry control. To enable only CONNECT BY use:

**Listing 4. Enabling CONNECT BY**

```
db2stop
db2set DB2_COMPATIBILITY_VECTOR=08
db2start
```
So, when should CONNECT BY be used and when should a recursive common table expressions be used?

Use CONNECT BY when:

- You need to return an ordered result set of an existing hierarchy or an email thread.
- There may be cycles that need to be caught or skipped.
- The recursion is over a self-referencing RI constraint.

Use recursive common table expressions when:

- You need to "generate rows" through recursion such as "all the business days of 2007."
- Ordering of the result set does not matter.
- You expect recursion of more than 64 levels.

**Date conversions**

DB2 UDB V8.1 introduced `TO_CHAR()` and `TO_DATE()` as synonyms for the generic `VARCHAR_FORMAT()` and `TIMESTAMP_FORMAT()` functions. However, these functions supported only a very narrow set of formats. In DB2 Viper 2, the coverage for formats has been greatly expanded. In essence, just about all formats commonly used in Oracle applications are supported unless they require National Language support (such as "Monday" vs. "Montag" or "December" vs. "Dezember"). Here is a list of the supported formats:

### Table 1. Supported formats

<table>
<thead>
<tr>
<th>Format element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>Century (00-99). If the last two digits of the four-digit year are zero, the result is the first two digits of the year. Otherwise, the result is the first two digits of the year plus one.</td>
</tr>
<tr>
<td>DD</td>
<td>Day of month (01-31).</td>
</tr>
<tr>
<td>DDD</td>
<td>Day of year (001-366).</td>
</tr>
<tr>
<td>FF[n]</td>
<td>Fractional seconds (0-999999). The number n is used to specify the number of digits to include in the returned value. Valid values for n are 1-6. The default is 6.</td>
</tr>
<tr>
<td>HH</td>
<td>HH behaves the same as HH12.</td>
</tr>
<tr>
<td>HH12</td>
<td>Hour of the day (01-12) in 12-hour format. AM is the default meridian indicator.</td>
</tr>
<tr>
<td>HH24</td>
<td>Hour of the day (00-24) in 24-hour format.</td>
</tr>
<tr>
<td>IW</td>
<td>ISO week of the year (01-53). The week starts on Monday and includes seven days. Week 1 is the first week of the year to contain a Thursday, which is equivalent to the first week of the year to contain January 4.</td>
</tr>
<tr>
<td>I</td>
<td>ISO year (0-9). The last digit of the year based on the ISO week that is returned.</td>
</tr>
<tr>
<td>IY</td>
<td>ISO year (00-99). The last two digits of the year based on the ISO week that is returned.</td>
</tr>
<tr>
<td>IYY</td>
<td>ISO year (000-999). The last three digits of the year based on the ISO week that is returned.</td>
</tr>
</tbody>
</table>
More details on the formats can be found in the DB2 Viper 2 Information Center (see the Resources section). For this article, some examples will suffice:

**Listing 5. Usage of TO_CHAR/TO_DATE**

```sql
SELECT CURRENT TIMESTAMP AS now,
    SUBSTR(TO_CHAR(CURRENT TIMESTAMP, format), 1, 25) AS formatted
FROM (VALUES ('YYYY/MM/DD'),
              ('DD.MM.YY'),
              ('MM-DD-YYYY'),
              ('YYYYMMDDHH24MISSNNNNN'),
              ('Q/Y')) AS T(format);
```

<table>
<thead>
<tr>
<th>NOW</th>
<th>FORMATTED</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007-07-12-18.19.30.784000</td>
<td>2007/07/12</td>
</tr>
<tr>
<td>2007-07-12-18.19.30.784000</td>
<td>12.07.07</td>
</tr>
<tr>
<td>2007-07-12-18.19.30.784000</td>
<td>20070712181930784000</td>
</tr>
<tr>
<td>2007-07-12-18.19.30.784000</td>
<td>2/07</td>
</tr>
</tbody>
</table>

5 record(s) selected.

```sql
SELECT formatted, TO_DATE(formatted, format) AS timestamp
FROM (VALUES('2007/07/12'          , 'YYYY/MM/DD'),
        ('12.07.07'            , 'DD.MM.YY'),
        ('07-12-2007'          , 'MM-DD-YYYY'),
        ('20070712181930784000', 'YYYYMMDDHH24MISSNNNNN'))
    AS T(formatted, format);
```

<table>
<thead>
<tr>
<th>FORMATTED</th>
<th>TIMESTAMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>2007/07/12</td>
<td>2007-07-12-00.00.00.000000</td>
</tr>
<tr>
<td>12.07.07</td>
<td>2007-07-12-00.00.00.000000</td>
</tr>
</tbody>
</table>

5 record(s) selected.
Limit result sets

Sometimes it is necessary to limit the number of rows returned by a query. Oracle applications typically utilize the `ROWNUM` pseudo column for that purpose. In DB2, there are two options to achieve the same effect:

- `FETCH FIRST n ROWS` clause
- `ROW_NUMBER() OLAP` function

To make porting of SQL using ROWNUM easier, three changes have been made:

1. `ROWNUM` is now supported as a synonym for `ROW_NUMBER() OVER()`. This captures all common usages of ROWNUM in the select list.
2. `ROW_NUMBER() OVER()` (and thus `ROWNUM`) can now be specified in the WHERE clause. This extension not only covers all the common usages of ROWNUM in Oracle, but also adds capabilities very close to the `LIMIT OFFSET` clause employed by some open source vendors because ROWNUM can be used together with a `BETWEEN` clause to allow easy result set pagination.
3. When a `ROW_NUMBER() OVER()` or `ROWNUM` is followed by an `ORDER BY` clause, the order is inherited into the OLAP function, guaranteeing that the values generated match the outer order.

Here are some example usages:

### Listing 6. Usage of ROWNUM

```sql
CREATE TABLE emp(id INT, name VARCHAR(20), salary DECIMAL(16));
INSERT INTO emp VALUES (1, 'Jones', 35000),
  (2, 'Newman', 42000),
  (3, 'Muller', 28500),
  (4, 'Green', 53800),
  (5, 'Kennedy', 47300),
  (6, 'Mason', 71000);

SELECT ROWNUM, name FROM emp ORDER BY salary;
1  -------------------
   Muller
   2  Jones
   3  Newman
   4  Kennedy
   5  Green
   6  Mason

6 record(s) selected.

SELECT name FROM emp WHERE ROWNUM < 3;
NAME
-------------
Jones
Newman
```
Presently, ROWNUM is under registry control. To enable only ROWNUM use:

**Listing 7. Enabling ROWNUM**

```
db2stop
db2set DB2_COMPATIBILITY_VECTOR=01
db2start
```

**Functions**

DB2 Viper 2 introduces a number of synonyms or new functions that make porting easier. These functions include:

- NVL
- DECODE
- LEAST and GREATEST
- BITAND, BITANDNOT, BITOR, BITXOR, and BITNOT

These functions are generally available with no need to set a registry variable.

**NVL**

NVL has been a precursor to COALESCE with the limitation that it can only handle two arguments. Its workings are quite simple. If the first argument NOT NULL NVL returns that argument, otherwise it returns the second argument. Therefore, NULL is returned only if both arguments are NULL. For simplicity in DB2 Viper 2, NVL has been made a synonym to COALESCE. Meaning that NVL accepts any number of arguments with a minimum of two.

**Listing 8. Usage of NVL()**

```
SELECT NVL(val1, val2)
FROM (VALUES (NULL,    1),
            (2   , NULL),
            (NULL, NULL),
            (4   ,    5)) AS T(val1, val2);
1
-----------
1
2

4 record(s) selected.
```
**DECODE**

Before the CASE expression, there was DECODE. DECODE, in a nutshell, is a function notation for what is called a *simple case expression*. All major vendors have long since added CASE expression support to their products. However, DECODE has retained a rather loyal following mainly because it is syntactically tight. There is one significant difference between CASE and DECODE though. In DECODE, NULLs are considered to be equal. The following is a comparison of CASE and DECODE.

**Listing 9. Usage of DECODE()**

```sql
SELECT CASE leftCol WHEN rightCol1 THEN 'First'
    WHEN rightCol2 THEN 'Second'
    ELSE 'else'
END AS case,
DECODE(leftCol, rightCol1, 'First', rightCol2, 'Second', 'else') AS decode
FROM (VALUES (5, 5, 6),
            (5, 6, 5),
            (5, 6, 7),
            (NULL, 5, NULL),
            (NULL, NULL, 5)) AS T(leftCol, rightCol1, rightCol2);
```

<table>
<thead>
<tr>
<th>CASE</th>
<th>DECODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>First</td>
<td>First</td>
</tr>
<tr>
<td>Second</td>
<td>Second</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>else</td>
<td>Second</td>
</tr>
<tr>
<td>else</td>
<td>First</td>
</tr>
</tbody>
</table>

5 record(s) selected.

**LEAST and GREATEST**

LEAST and GREATEST return the smallest or biggest value of a set of arguments. Alternative names for these functions are the MIN and MAX scalar functions, which must not be confused with the aggregate functions of the same name. While MIN/MAX with one argument aggregate values from a set of rows and return the smallest or biggest value respectively, MIN (aka LEAST) and MAX (aka GREATEST) with two or more arguments operate only on their input arguments for the current row with no grouping effect. Aside from this principle difference, LEAST and GREATEST also return a NULL if any argument is NULL, where the MIN and MAX aggregate functions ignore NULLs. The following is an example comparing the functions.

**Listing 10. Usage of LEAST() and GREATEST()**

```sql
SELECT LEAST(val1, val2, val3) AS least,
GREATEST(val1, val2, val3) AS greatest,
MIN(val1, val2, val3) AS min,
MAX(val1, val2, val3) AS max
FROM (VALUES (3,    2, 1),
            (2, NULL, 2),
            (1,    2, 3)) AS T(val1, val2, val3);
```

<table>
<thead>
<tr>
<th>LEAST</th>
<th>GREATEST</th>
<th>MIN</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

3 record(s) selected.
Bit manipulation

DB2 Viper 2 introduces a set of functions that are used to efficiently encode, decode, and test bit arrays. Simply put, the bit manipulation functions view a whole number in its binary two's complement representation and allow the setting, resetting, toggling, or testing of individual bits. Note that the binary representation is independent of the internal representation. Meaning it is independent of endianness or the encoding of the base type, such as DECFLOAT. The number of bits supported depends on the datatypes used:

Table 2. Number of bits supported for a data type

<table>
<thead>
<tr>
<th>Number of bits</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>SMALLINT</td>
</tr>
<tr>
<td>32</td>
<td>INTEGER</td>
</tr>
<tr>
<td>64</td>
<td>BIGINT</td>
</tr>
<tr>
<td>113</td>
<td>DECFLOAT(34)</td>
</tr>
</tbody>
</table>

The meaning of each function and its usage is rather straight forward. Nonetheless, Table 3 explains each function followed by some examples.

Table 3. Bit operation functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BITAND(&lt;expr1&gt;, &lt;expr2&gt;)</td>
<td>Returns a whole number where each bit in the binary representation is set if that bit was set in both arguments. This function is typically used to test whether bits are set.</td>
</tr>
<tr>
<td>BITOR(&lt;expr1&gt;, &lt;expr2&gt;)</td>
<td>Returns a whole number where each bit is set if that bit is set in either of the two arguments. This function is typically used to set bits.</td>
</tr>
<tr>
<td>BITNOT(&lt;expr&gt;)</td>
<td>This function inverts all bits of the arguments. Since DB2 uses the two's complement that implies that the inverse of 0 is -1. This translates into all the bits supported by the data type being set.</td>
</tr>
<tr>
<td>BITANDNOT(&lt;expr1&gt;, &lt;expr2&gt;)</td>
<td>This function is a short form for BITAND(&lt;expr1&gt;, BITNOT(&lt;expr2&gt;)). The function unsets all bits in the first argument that are set in the second argument. So its common usage is to reset bits.</td>
</tr>
<tr>
<td>BXITXOR(&lt;expr1&gt;, &lt;expr2&gt;)</td>
<td>XOR stands for eXclusive OR. This means that the function returns a number for which bits are set, if they are set only in one of the two arguments. If a bit is set or unset in both arguments the resulting bit is unset. The common usage for this function is to flip bits.</td>
</tr>
</tbody>
</table>

In general, you should be aware of the data types used as arguments for these functions. If the types do not match up, DB2 goes through regular type promotion from one type to the other.
Meaning a positive value gets padded with a number of binary zeros to the left while a negative value is padded to the left with binary ones. So using a consistent type is strongly advised. Also, be aware that while DECFLOAT supports 113 bits, two BIGINTS that take the same amount of storage support 128 bits. The price of convenience.

Listing 11 provides code examples:

**Listing 11. Usage of BIT manipulation functions**

```sql
--#SET TERMINATOR @
CREATE FUNCTION printbits(value INTEGER) RETURNS VARCHAR(16)
BEGIN ATOMIC
  DECLARE res VARCHAR(16) DEFAULT '';
  DECLARE bit INTEGER DEFAULT 1;
  WHILE bit < 32769 DO
    SET res = res || CASE WHEN BITAND(value, bit) = bit THEN '1' ELSE '0' END,
                        bit = bit * 2;
  END WHILE;
  RETURN res;
END
--#SET TERMINATOR ;
SELECT printbits(value)                 AS binaryvalue,
     printbits(BITAND(value, bit))    AS bitand,
     printbits(BITXOR(value, bit))    AS bitxor,
     printbits(BITANDNOT(value, bit)) AS bitandnot,
     printbits(BITOR(value, bit))     AS bitor,
     CASE WHEN BITAND(value, bit) = bit
            THEN '1' ELSE '0' END       AS "bit"
FROM (VALUES (SMALLINT(24457)))       AS value(value),
       (VALUES (    1),
              (    2),
              (    4),
              (    8),
              (   16),
              (   32),
              (   64),
              (  128),
              (  256),
              (  512),
              ( 1024),
              ( 2048),
              ( 4096),
              ( 8192),
              (16384),
              (32768)) AS bit(bit);
```

<table>
<thead>
<tr>
<th>BINARYVALUE</th>
<th>BITAND</th>
<th>BITXOR</th>
<th>BITANDNOT</th>
<th>BITOR</th>
<th>BIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>10010001111111010</td>
<td>1000000000000000</td>
<td>0010001111111010</td>
<td>00010001111111010</td>
<td>10010001111111010</td>
<td>1</td>
</tr>
<tr>
<td>10010001111111010</td>
<td>0000000000000000</td>
<td>1010001111111010</td>
<td>10010001111111010</td>
<td>10100011111111010</td>
<td>0</td>
</tr>
<tr>
<td>10010001111111010</td>
<td>0000000000000000</td>
<td>1010001111111010</td>
<td>10010001111111010</td>
<td>10100011111111010</td>
<td>0</td>
</tr>
<tr>
<td>10010001111111010</td>
<td>0000000000000000</td>
<td>1010001111111010</td>
<td>10010001111111010</td>
<td>10100011111111010</td>
<td>0</td>
</tr>
<tr>
<td>10010001111111010</td>
<td>0000000000000000</td>
<td>1010001111111010</td>
<td>10010001111111010</td>
<td>10100011111111010</td>
<td>0</td>
</tr>
<tr>
<td>10010001111111010</td>
<td>0000000000000000</td>
<td>1010001111111010</td>
<td>10010001111111010</td>
<td>10100011111111010</td>
<td>0</td>
</tr>
<tr>
<td>10010001111111010</td>
<td>0000000000000000</td>
<td>1010001111111010</td>
<td>10010001111111010</td>
<td>10100011111111010</td>
<td>0</td>
</tr>
<tr>
<td>10010001111111010</td>
<td>0000000000000000</td>
<td>1010001111111010</td>
<td>10010001111111010</td>
<td>10100011111111010</td>
<td>0</td>
</tr>
<tr>
<td>10010001111111010</td>
<td>0000000000000000</td>
<td>1010001111111010</td>
<td>10010001111111010</td>
<td>10100011111111010</td>
<td>0</td>
</tr>
<tr>
<td>10010001111111010</td>
<td>0000000000000000</td>
<td>1010001111111010</td>
<td>10010001111111010</td>
<td>10100011111111010</td>
<td>0</td>
</tr>
<tr>
<td>10010001111111010</td>
<td>0000000000000000</td>
<td>1010001111111010</td>
<td>10010001111111010</td>
<td>10100011111111010</td>
<td>0</td>
</tr>
<tr>
<td>10010001111111010</td>
<td>0000000000000000</td>
<td>1010001111111010</td>
<td>10010001111111010</td>
<td>10100011111111010</td>
<td>0</td>
</tr>
<tr>
<td>10010001111111010</td>
<td>0000000000000000</td>
<td>1010001111111010</td>
<td>10010001111111010</td>
<td>10100011111111010</td>
<td>0</td>
</tr>
</tbody>
</table>
16 record(s) selected.

```
SELEX value,     
  hex(value) AS INTERNAL,  
  printbits(value) AS BINARY,  
  printbits(BITNOT(value)) AS BITNOT  
FROM (VALUES(SMALLINT(24457))) AS value(value);  
VALUE  INTERNAL BINARY           BITNOT  
------ -------- ---------------- ----------------  
24457 895F 1001000111111010 0110111000000101  
```

1 record(s) selected.

### (+) outer join syntax

Before supporting the SQL standard OUTER JOIN syntax, Oracle used a proprietary syntax to model outer joins using an extension to the WHERE clause. While this syntax is long since deprecated, there are still many old applications that use this syntax. To make matters worse, plenty of developers have not adopted the SQL standard syntax for various reasons and continue to produce new applications based on the old syntax.

To support porting of those old applications (+), style join syntax has been added to DB2 Viper 2. DB2 development strongly discourages usage of this feature for any other purpose. Therefore, (+) join syntax is under registry control. To enable only (+) join syntax use:

**Listing 12. Enabling (+) join syntax**

```
db2stop
db2set DB2_COMPATIBILITY_VECTOR=04
db2start
```

Having given this disclaimer, the syntax will now briefly be introduced, assuming that those who need to know, already know, and merely need confirmation that support is available.

Simply put, the (+) join syntax uses the implicit join syntax where tables are listed in the FROM clause and predicates correlate the tables in the WHERE clause. Do denote that a predicate in the WHERE clause is an outer join. All column references to the inner (that is the null producing side) are trailed by a "(+)". If the predicate references more than one column, then all columns must be marked in the same way. Full outer joins are not supported. Also "bushy" joins, that is joins which would require the setting of braces in the SQL standard, are not supported. Anyway, this feature has already been given more attention than it deserves.

Listing 13 provides a couple of examples comparing (+) notation and SQL standard outer joins.

**Listing 13. (+) join examples**

```
CREATE TABLE emp(empid   INT,  
  emppname VARCHAR(10),  
  deptid    INT);  
CREATE TABLE dept(deptid   INT,  
  deptname VARCHAR(10),  
  mgrid INTEGER);  
INSERT INTO emp VALUES (1, 'Jones', 10),
(2, 'Meyer', 10),
(3, 'Newman', 20),
(4, 'Feldman', NULL),
(5, 'Hedges', NULL);

INSERT INTO dept VALUES (10, 'Sales', 4),
(20, 'Marketing', 5);

SELECT empname, deptname
FROM emp LEFT OUTER JOIN dept ON emp.deptid = dept.deptid;

EMPNAME  DEPTNAME
--------- ----------
Jones     Sales
Meyer     Sales
Newman    Marketing
Feldman   -
Hedges    -

5 record(s) selected.

SELECT empname, deptname
FROM emp, dept WHERE emp.deptid = dept.deptid (+);

EMPNAME  DEPTNAME
--------- ----------
Jones     Sales
Meyer     Sales
Newman    Marketing
Feldman   -
Hedges    -

5 record(s) selected.

SELECT emp.empname, mgr.empname AS mgrname
FROM emp LEFT OUTER JOIN dept ON emp.deptid = dept.deptid
LEFT OUTER JOIN emp AS mgr ON dept.mgrid = mgr.empid;

EMPNAME   MGRNAME
---------- ----------
Jones      Feldman
Meyer      Feldman
Newman     Hedges
Feldman    -
Hedges     -

5 record(s) selected.

SELECT emp.empname, mgr.empname AS mgrname
FROM emp, dept, emp as mgr
WHERE emp.deptid = dept.deptid (+)
AND dept.mgrid = mgr.empid (+);

EMPNAME   MGRNAME
---------- ----------
Jones      Feldman
Meyer      Feldman
Newman     Hedges
Feldman    -
Hedges     -

5 record(s) selected.

**Miscellaneous**

Often the differences between SQL dialects are annoyingly trivial. Meaning that it can be incomprehensible for the developer why identical behavior uses different syntax, often without leaving any option for compatible SQL code. The items in this section are geared to decrease these nuisance differences.
DUAL

Sometimes all that is required of the data server is the computation of a simple scalar expression, or the retrieval of a built-in variable, such as the current time. However, since SQL is table centric such a task is not nearly as trivial and consistently solved as one might expect. One possible solution is to provide a dummy table that has one row and one column, and can serve up the environment to perform the computation. In DB2, this dummy table is called SYSIBM.SYSDUMMY1. In Oracle, it is called DUAL. So in DB2 Viper 2, you can also use DUAL. Note that DUAL does not require a schema. If the functionality is enabled, any unqualified table reference named "DUAL" is presumed to be this one. So it is best not to use DUAL in your regular schema. Listing 14 shows a quick example:

**Listing 14. DUAL and its DB2 alternative**

```sql
SELECT CURRENT TIME FROM DUAL;
1
--------
14:20:36
1 record(s) selected.

SELECT CURRENT TIME FROM SYSIBM.SYSDUMMY1;
1
--------
14:21:15
1 record(s) selected.

VALUES CURRENT TIME;
1
--------
14:21:33
1 record(s) selected.

SELECT * FROM DUAL;
DUMMY
-----
X
1 record(s) selected.
```

Due to the potential conflict with existing tables, DUAL has been placed under registry control. To enable only DUAL, use:

**Listing 15. Enabling DUAL**

```sql
db2stop
db2set DB2_COMPATIBILITY_VECTOR=02
db2start
```

CURRVAL and NEXTVAL

When sequences were introduced in DB2 V7.2, there were three changes that had to be made from the precedent to make the feature palatable to the SQL standard:

- Separation of namespaces between tables, routines, and sequences
- Clear separation of the previous value from the next value
• Usage of an expression instead of a pseudo column

As a result, any sequence generation and sequence look-up is incompatible between Oracle and DB2. DB2 Viper 2 adds the following toleration:

• Allow pseudo column notation to retrieve or generate a sequence value. Note that pseudo columns are only resolved after regular name resolution fails.
• Allow CURRVAL as notation to retrieve the previous value in pseudo column notation.

Note that the second bullet does not imply that a CURRVAL in the same select list as a NEXTVAL returns NEXTVAL, as it does in Oracle. DB2 maintains that separation. Listing 16 shows examples of the new support:

Listing 16. CURRVAL and NEXTVAL usage

```
CREATE SEQUENCE seq;

SELECT NEXT VALUE FOR seq AS DB2NEXT, seq.NEXTVAL AS ORANEXT
  FROM SYSIBM.SYSDUMMY1;
DB2NEXT    ORANEXT
----------- -----------
1           1
1 record(s) selected.

SELECT PREVIOUS VALUE FOR seq AS DB2PREV, seq.CURRVAL AS ORAPREV
  FROM SYSIBM.SYSDUMMY1;
DB2PREV    ORAPREV
----------- -----------
1           1
1 record(s) selected.

SELECT seq.CURRVAL AS CURRVAL, seq.NEXTVAL AS NEXTVAL
  FROM SYSIBM.SYSDUMMY1;
CURRVAL    NEXTVAL
----------- -----------
1           2
1 record(s) selected.
```

Inline views

DB2 users know inline views as nested subqueries. In other words, they are SELECTs in the FROM clause. In DB2, an inline view traditionally must be named. In Oracle it must not be named. The conundrum to a SQL developer is self evident. Only in recent releases of Oracle, can complex queries be written using common table expressions (also known as the WITH clause), and also be run against DB2. DB2 Viper 2 makes naming of nested subqueries optional, allowing for shared SQL for complex queries. DB2 users will appreciate that omitting a name exposes the function name of a table function. Listing 17 provides a simple example illustrating the behavior:

Listing 17. Inline view usage

```
SELECT * FROM (VALUES (1), (2), (3)) AS T(c1); C1
----------
1
2
3
```
MINUS
When subtracting one query from another, DB2 and Oracle have also chosen different paths. In DB2, the EXCEPT keyword is used. In Oracle, the keyword is MINUS. DB2 Viper 2 now accepts MINUS as a synonym for EXCEPT. So the two following queries are identical:

Listing 18. Usage of MINUS
CREATE TABLE T1(c1 INT);
CREATE TABLE T2(c1 INT);
INSERT INTO T1 VALUES (1), (2), (3), (4);
INSERT INTO T2 VALUES (1), (4);
SELECT * FROM T1
EXCEPT
SELECT * FROM T2;
C1
---------
1
2
3
2 record(s) selected.

SELECT * FROM T1
MINUS
SELECT * FROM T2;
C1
---------
1
2
3
2 record(s) selected.

SELECT UNIQUE
While Oracle has long since adopted the DISTINCT keyword to filter out duplicate rows, there are still applications that use UNIQUE as a synonym for DISTINCT. Furthermore, UNIQUE is also found in Informix IDS where it can be used in some additional places. DB2 Viper 2 makes UNIQUE a synonym for the DISTINCT keyword anywhere except for CREATE DISTINCT TYPE. Listing 19 shows its usage:
Listing 19. Usage of UNIQUE

```sql
SELECT UNIQUE * FROM (VALUES (1), (2), (2), (3));

+----+
|    |
+----+
| 1  |
| 2  |
| 3  |
+----+
3 record(s) selected.

SELECT DISTINCT * FROM (VALUES (1), (2), (2), (3));

+----+
|    |
+----+
| 1  |
| 2  |
| 3  |
+----+
3 record(s) selected.
```

Related features

Besides the compatibility features highlighted in the preceding sections, DB2 Viper 2 sports a host of other major new functionality. Some of them can make porting easier and deserve an honorable mention here and in dedicated articles in this forum.

Optimistic locking

DB2 Viper 2's support for optimistic locking strategies includes two core pieces:

1. Automatic time stamping or change marking of rows as they are modified in the database.
2. Two new functions `RID()` and `RID_BIT()` that externalize the physical position of a row.

While some may disapprove of exposing physical properties of a row through SQL, it is undeniable that `RID()` and `RID_BIT()` make porting of applications that use `ROWID` a lot easier.

Array processing

The SQL standard ARRAY data type is supported in DB2 Viper 2 within SQL procedures as well as for procedure parameters. This feature allows for more efficient movement of data sets from the application to the DB2 server. It enables simplified porting of not only `VARRAY` types, but also of `FORALL` and `BULK COLLECT` constructs through array aggregation using the `ARRAY_AGG()` function, as well as unnesting of arrays into tables using the `UNNEST()` operator.

Global session variable support

DB2 Viper 2 extends SQL with a new data object called `global session variable`. Global session variables have a schema qualified, persistent definitions in the catalog, but their content is private to any given session. Global session variables facilitate porting of `package variables` when mapping a `package` to a DB2 schema.

Database roles

Traditionally, DB2 uses operating system facilities to authenticate users and verify group memberships. There is no user information or passwords stored in the DBMS. This has not
changed and is unlikely to in the future. However, there is value in managing group memberships within the data server. In a nutshell, groups managed within the data server are called roles. Roles are supported by DB2 Viper 2. This simplifies porting of applications that rely on roles.

**Decimal floating point type**

There has long been a need for a numeric data type that provides both decimal accuracy as well as floating point behavior. However, that need is not limited to the database. Instead it covers programming languages, databases, and ultimately hardware. After all, what good is a type that is slow to do arithmetic on? There are few, if any, companies in the world who have the breadth and patience to rise to such a broad challenge these days. Over the past years, IBM has quietly worked to introduce a true standardized decimal floating point into C, Java, its new Power 6 processors, DB2 9 for zOS, and now DB2 Viper 2. In SQL, this new type is called **DECFLOAT** and presently has an accuracy of 16 or 34 digits at a range beyond 10 to the power of 6000. DB2 Viper 2 exploits the Power 6 hardware acceleration. I am confident other hardware vendors will follow suit.

**DECFLOAT** support greatly simplifies porting of the proprietary (and software emulated) **NUMBER** data type.

**Conclusion**

When enabling DB2 for your application, the cost of changing and maintaining your code is a major cost factor. Any difference between SQL dialects has to be found and worked around increasing the time it takes to do the port as well as complicating testing. DB2 Viper 2 provides significant enhancements that help you drive this cost down, making a port more straight forward.

As DB2 Viper 2 nears release, you may find additional enhancements with the same thrust. The goal is that all these enhancements are exploited by the migration toolkit, increasing its conversion rate as well as overall performance of converted application.

**Appendix**

Throughout this article there has been reference to the **DB2_COMPATIBILITY_VECTOR** registry variable. The purpose of this variable is to enable or disable some of the features discussed. To switch on a feature, you must add its bit to the current setting of the register. To switch a feature off, the respective bit needs to be unset. After setting the variable, you need to restart DB2. The settings are defined as follow:

<table>
<thead>
<tr>
<th>Bit position</th>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (0x01)</td>
<td>ROWNUM</td>
<td>Enable ROWNUM pseudo column in SELECT list and WHERE clause.</td>
</tr>
<tr>
<td>2 (0x02)</td>
<td>DUAL</td>
<td>Enable the DUAL dummy table</td>
</tr>
<tr>
<td>3 (0x04)</td>
<td>(+) join operator</td>
<td>Enable Oracle style outer join syntax</td>
</tr>
<tr>
<td>4 (0x08)</td>
<td>Hierarchical queries</td>
<td>Enable CONNECT BY syntax for recursion</td>
</tr>
</tbody>
</table>
For example, to enable both **ROWNUM** (0x01) and **DUAL** (0x02), use:

**Listing 20. Enabling ROWNUM and DUAL**

```
db2stop
db2set DB2_COMPATIBILITY_VECTOR=03
db2start
```

To add hierarchical query support, or 0x03 with 0x08 which is 0x0B, and enter:

**Listing 21. Enabling ROWNUM, DUAL, and CONNET BY**

```
db2stop
db2set DB2_COMPATIBILITY_VECTOR=0B
db2start
```