

# Analytic Functions in Oracle Warehouse Builder

*An Oracle White Paper  
August 2004*

# Optimizing Performance with Warehouse Builder

Executive Summary .....	3
Introduction .....	4
Code generation and analytic functions .....	5
Example 1 .....	8
Example 2 .....	14
Restrictions .....	15
Future direction .....	16
Conclusion.....	17

## **Executive Summary**

Analytic Functions are an important feature of the Oracle database that allows the users to enhance SQL's analytical processing capabilities. These functions enable the user to calculate rankings and percentiles, moving window calculations, lag/lead analysis, top-bottom analysis, linear regression analytics and other similar calculation-intensive data processing.

This whitepaper describes how to use some of these analytical functions in the ETL (extraction, transformation and loading) processing implementation using Oracle Warehouse Builder.

## **Introduction**

Currently analytic functions in OWB are not directly supported because of the way the code is generated for the row-based operational mode. In the following article, a workaround that enables users to use a number of analytic functions in Warehouse Builder is described.

## Code generation and analytic functions

There are two main operational modes in Warehouse Builder mappings – set based and row based (the remaining three modes – set-based, fail over to row-based, set-based, fail over to row-based (target only) and row based (target only) are variations of the first two). Every time PL/SQL code is generated from a mapping design, the resulting package will contain code for all five operating modes, and when the package is deployed, the code for all the modes must be correct.

Because of the way the code is generated in the row based mode, a cursor that will encompass all the objects of the mapping graph up to the point where an SQL-only operator is encountered will be generated (an SQL-only operator is an operator that must process the data flow in one go, not row-by-row; such operators are, see below a list of SQL-only operators). This cursor is referred here as the SQL-context. The cursor will then be opened and values from this cursor will be fetched and assigned to PL/SQL variables for further processing. This is referred as PL/SQL context. Therefore, the code generator identifies a cut-off operator that represents a boundary between the SQL and the PL/SQL context.

Analytic functions can be enclosed in Warehouse Builder expressions. Following the above considerations, it is important to note that analytic functions can only be used within an SQL statement (i.e. within the above mentioned SQL context). If an analytic function such as ROW\_NUMBER is used in a PL/SQL context, ("x := row\_number() ..."), there will be compilation errors during the deployment phase of the Warehouse Builder generated code. However, OWB users have ways to influence the placement of expressions containing analytic functions in order to prevent the generation of analytic functions in a PL/SQL context. In generating the row-based code, OWB first attempts to identify an operator that will be used as a cut-off point for the creation of a cursor (SQL context). Once this operator is identified, all the upstream (closer to the source) operators, including the cut-off operator, will be used to generate the cursor definition statement (SQL context). All the remaining downstream (closer to the target) operators will be implemented using pure PL/SQL constructs. Because the cursor definition statement is in SQL context, which accepts analytic functions (while they are not allowed in PL/SQL constructs), by placing the expression containing the analytic function before the cut-off operator (within the SQL context), the generated code will be valid.

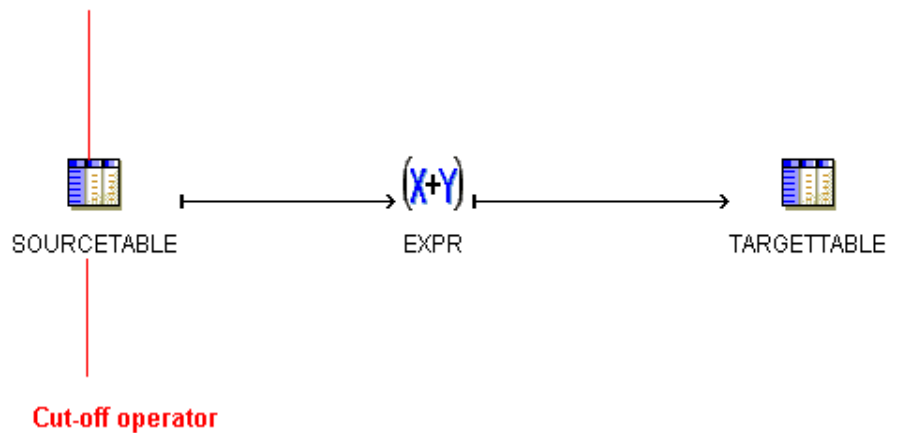
OWB identifies the cursor-defining (cut-off) operator using the following algorithm:

1. If the operating mode is "row-based (target only)" or "set-based failover to row-based (target only)", then the cursor-

defining operator is the operator immediately upstream of the target operator.

2. If the operating mode is "row-based" or "set-based failover to row-based", then the cursor-defining operator is the last SQL-only operator, not including the target operator.

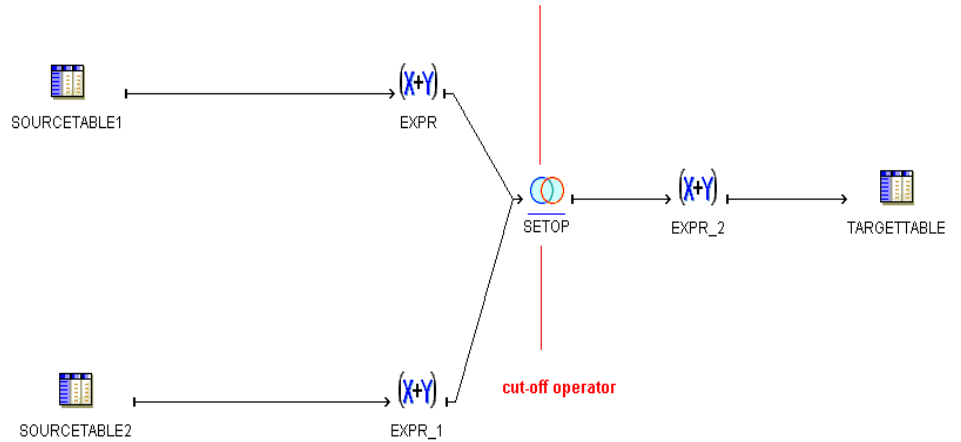
For example, given the algorithm above, in the following mapping:



**Figure 1. Simple mapping**

the cursor will be created at source table node (cut-off point) because it is SQL-only; Expression is not SQL-only, while the target table is SQL-only but since it is the target of the mapping, it is excluded from consideration.

In another example,



**Figure 2. Coupled mapping**

The Set operator is the cursor-defining (cut-off) node.

Knowing how OWB identifies the cursor-defining operator is important since OWB users can take advantage of it to force expressions containing analytic functions be generated within cursors. If an SQL-only operator is inserted downstream (closer to the target) of the analytic functions, all will be fine.

The following operators are Warehouse Builder's SQL-only operators:

- Tables
- Views
- Sequences
- Materialized views
- Dimensions
- Cubes
- Joins
- Set operators
- Aggregation operators
- Sort operators
- De-duplicators

## Example 1

In the example below, provided by Óscar Javier Blanco Huerga from the Oficina de Cooperación Universitaria in Madrid, Spain, a dummy de-duplicator operator is used to successfully include an analytic function in the mapping.

A simple mapping uses a ROW\_NUMBER analytic function to rank the sales by product and region every month. By using this rank flag it will be much easier for the reporting tool to produce top/bottom reports on product sales in various sales regions by months (best selling product by month and region, worst selling product by month and region etc.). The assumption is that sales data is already summarized by product, region and month. For example if the data in the SALES table is:

```
SQL> select * from sales;
```

PRODUCT_ID	REGION	SALES_AMOUNT	SALE_MONTH
1	SO	1000	1
2	SO	1100	1
3	SO	900	1
2	NO	1900	1
3	NO	1700	1
1	NO	1600	1
1	SO	1300	2
2	SO	1050	2
3	SO	1400	2
2	NO	1850	2
3	NO	2200	2
1	NO	1900	2

The data in the RANKED\_SALES table will contain the REGION\_RANK flag that is ranking products by sale for every region and month combination. Reporting tools can then use this flag can for top/bottom analysis:

SQL> select \* from ranked\_sales;

PRODUCT_ID	REGION	SALES_AMOUNT	SALE_MONTH	REGION_RANK
1	NO	1600	1	3
3	NO	1700	1	2
2	NO	1900	1	1
2	NO	1850	2	3
1	NO	1900	2	2
3	NO	2200	2	1
3	SO	900	1	3
1	SO	1000	1	2
2	SO	1100	1	1
2	SO	1050	2	3
1	SO	1300	2	2
3	SO	1400	2	1

We start with the straight mapping that will produce the un-deployable code:

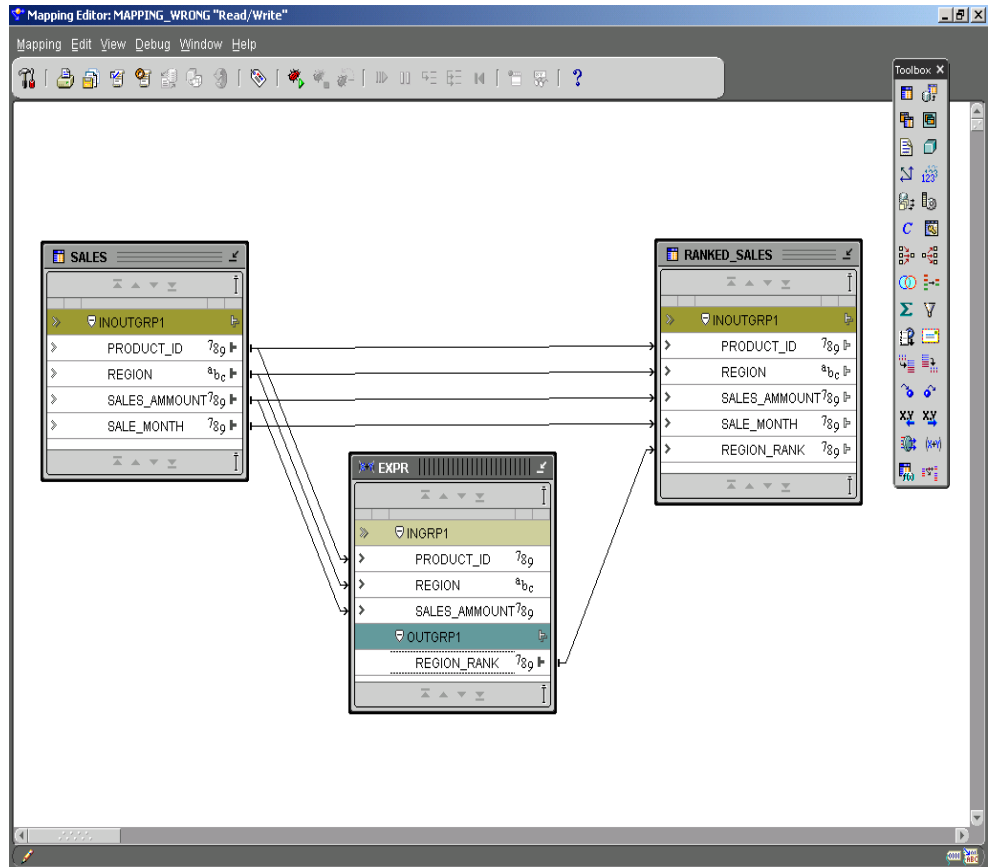
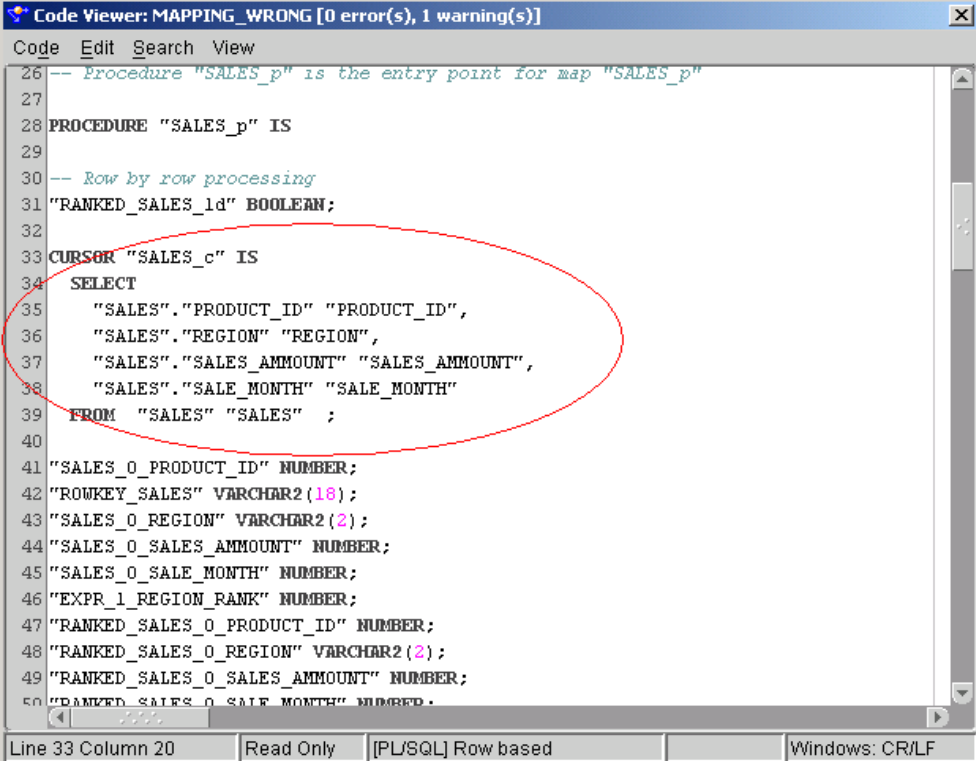


Figure 3. Incorrect mapping

The REGION\_RANK output column of the expression EXPR is calculated as:

```
ROW_NUMBER() OVER (PARTITION BY INGRP1.REGION,  
                    INGRP1.SALE_MONTH  
                    ORDER BY INGRP1.SALES_AMOUNT DESC)
```

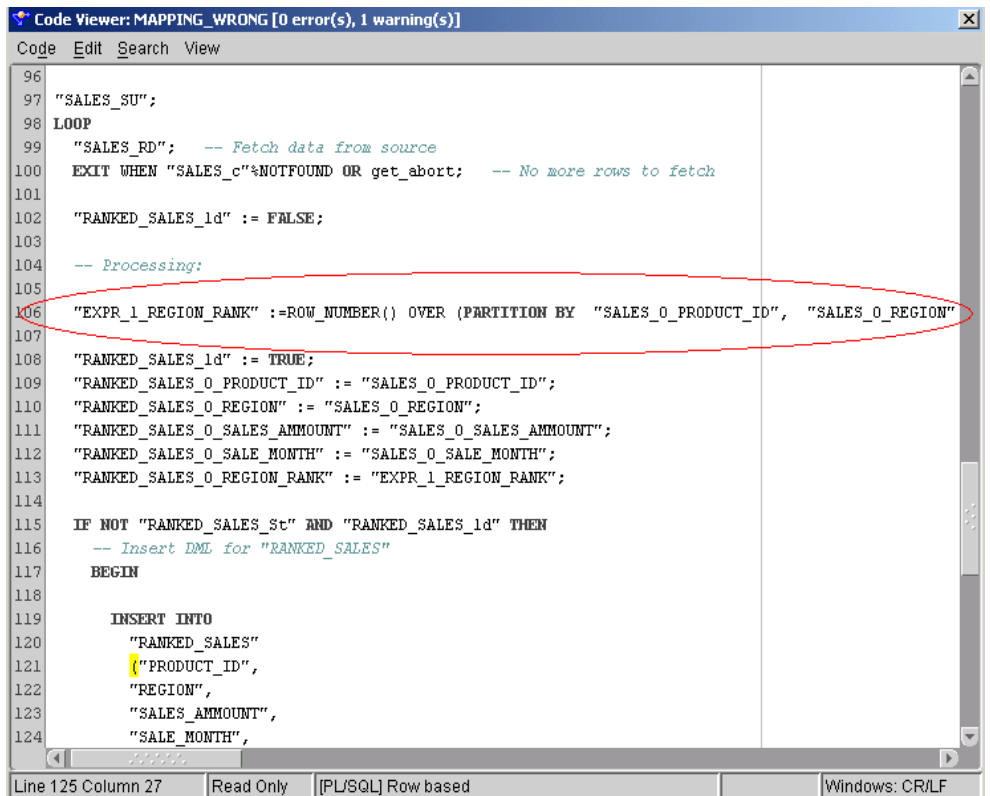
When the row-based generated code is inspected, we can see that a cursor that selects rows from the SALES table only will first be generated (since there are no SQL-only operators in the mapping after the expression containing the analytic function):



```
Code Viewer: MAPPING_WRONG [0 error(s), 1 warning(s)]
Code Edit Search View
26 -- Procedure "SALES_p" is the entry point for map "SALES_p"
27
28 PROCEDURE "SALES_p" IS
29
30 -- Row by row processing
31 "RANKED_SALES_ld" BOOLEAN;
32
33 CURSOR "SALES_c" IS
34 SELECT
35   "SALES"."PRODUCT_ID" "PRODUCT_ID",
36   "SALES"."REGION" "REGION",
37   "SALES"."SALES_AMMOUNT" "SALES_AMMOUNT",
38   "SALES"."SALE_MONTH" "SALE_MONTH"
39 FROM "SALES" "SALES" ;
40
41 "SALES_0_PRODUCT_ID" NUMBER;
42 "ROWKEY_SALES" VARCHAR2(18);
43 "SALES_0_REGION" VARCHAR2(2);
44 "SALES_0_SALES_AMMOUNT" NUMBER;
45 "SALES_0_SALE_MONTH" NUMBER;
46 "EXPR_1_REGION_RANK" NUMBER;
47 "RANKED_SALES_0_PRODUCT_ID" NUMBER;
48 "RANKED_SALES_0_REGION" VARCHAR2(2);
49 "RANKED_SALES_0_SALES_AMMOUNT" NUMBER;
50 "RANKED_SALES_0_SALE_MONTH" NUMBER;
```

**Figure 4. Incorrect cursor**

Then, after the cursor is opened, the return value of the analytic function will be assigned to a variable, and this will cause the deployment (compilation) error making the code un-deployable:

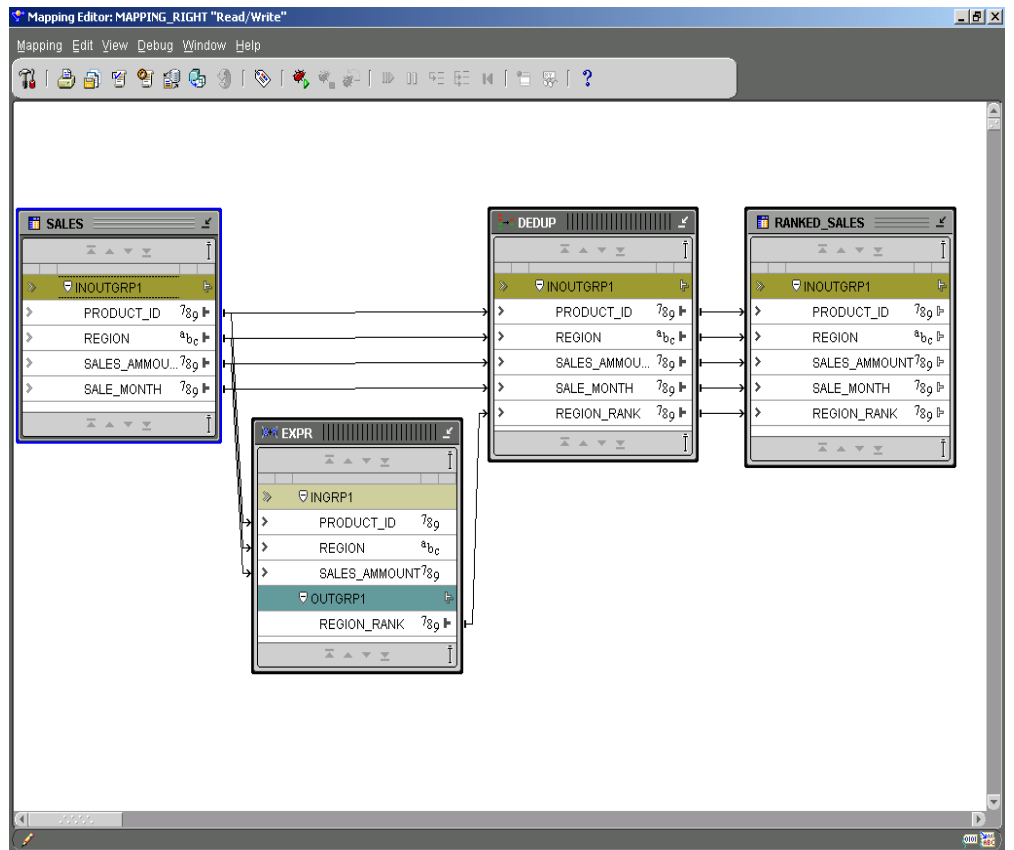


```
Code Viewer: MAPPING_WRONG [0 error(s), 1 warning(s)]
Code Edit Search View
96
97 "SALES_SU";
98 LOOP
99 "SALES_RD"; -- Fetch data from source
100 EXIT WHEN "SALES_c"%NOTFOUND OR get_abort; -- No more rows to fetch
101
102 "RANKED_SALES_ld" := FALSE;
103
104 -- Processing:
105
106 "EXPR_1_REGION_RANK" :=ROW_NUMBER() OVER (PARTITION BY "SALES_0_PRODUCT_ID", "SALES_0_REGION"
107
108 "RANKED_SALES_ld" := TRUE;
109 "RANKED_SALES_0_PRODUCT_ID" := "SALES_0_PRODUCT_ID";
110 "RANKED_SALES_0_REGION" := "SALES_0_REGION";
111 "RANKED_SALES_0_SALES_AMMOUNT" := "SALES_0_SALES_AMMOUNT";
112 "RANKED_SALES_0_SALE_MONTH" := "SALES_0_SALE_MONTH";
113 "RANKED_SALES_0_REGION_RANK" := "EXPR_1_REGION_RANK";
114
115 IF NOT "RANKED_SALES_st" AND "RANKED_SALES_ld" THEN
116 -- Insert DML for "RANKED_SALES"
117 BEGIN
118
119 INSERT INTO
120 "RANKED_SALES"
121 ("PRODUCT_ID",
122 "REGION",
123 "SALES_AMMOUNT",
124 "SALE_MONTH",
```

Line 125 Column 27 | Read Only | [PL/SQL] Row based | Windows: CR/LF

**Figure 5. Code causing the error**

If, on the other hand, an SQL-only operator, such as a dummy de-duplicator is added to the mapping after the expression (dummy because we assume there will be no duplicate rows here and the mapping logic will not be altered by the de-duplicator), the cursor generation for the row-based code is moved from the table to the next SQL-only operator (the de-duplicator). In other words, the addition of the de-duplicator after the expression containing the analytic function...



**Figure 6. Correct map**

... will cause the row-based cursor to include the analytic function...

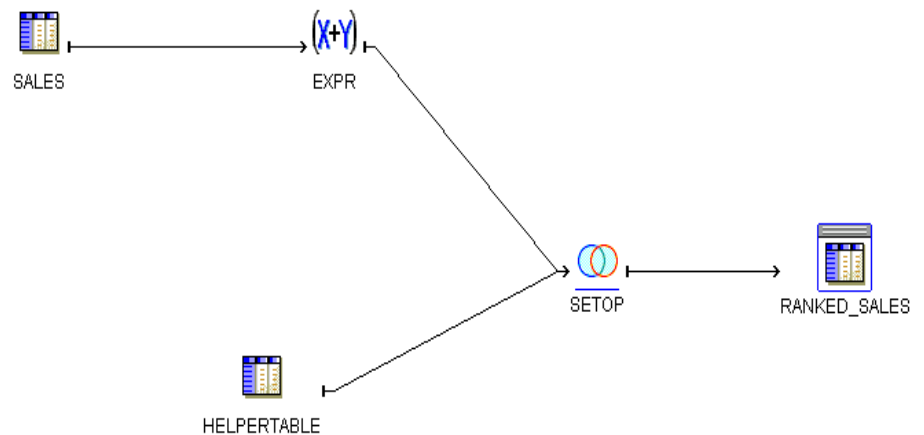
```
Code Viewer: MAPPING_RIGHT [0 error(s), 1 warning(s)]
Code Edit Search View
32
33 CURSOR "DEDUP_c" IS
34 SELECT
35     "DEDUP"."PRODUCT_ID" "PRODUCT_ID",
36     "DEDUP"."REGION" "REGION",
37     "DEDUP"."SALES_AMMOUNT" "SALES_AMMOUNT",
38     "DEDUP"."SALE_MONTH" "SALE_MONTH",
39     "DEDUP"."REGION_RANK" "REGION_RANK"
40 FROM (SELECT
41     DISTINCT
42     "SALES"."PRODUCT_ID" "PRODUCT_ID",
43     "SALES"."REGION" "REGION",
44     "SALES"."SALES_AMMOUNT" "SALES_AMMOUNT",
45     "SALES"."SALE_MONTH" "SALE_MONTH",
46     (ROW_NUMBER() OVER (PARTITION BY "SALES"."PRODUCT_ID", "SALES"."REGION" ORDER BY "SALES"."SALES_AMMOUNT"
47     FROM "SALES"."SALES" ) "DEDUP" ;
48
49 "DEDUP_0_PRODUCT_ID" NUMBER;
50 "DEDUP_0_REGION" VARCHAR2(2);
51 "ROWKEY_DEDUP" VARCHAR2(18);
52 "DEDUP_0_SALES_AMMOUNT" NUMBER;
53 "DEDUP_0_SALE_MONTH" NUMBER;
54 "DEDUP_0_REGION_RANK" NUMBER;
55 "RANKED_SALES_0_PRODUCT_ID" NUMBER;
```

**Figure 7. Correct code**

...and the code is now deployable.

## Example 2

If the de-duplicator solution from the previous example is not applicable because the distinct result is not desirable, it is possible to use the set operation with an empty helper table:



**Figure 8. An alternative solution**

Here the set operator is set to "Union All" and the Helper Table is a dummy table that is always empty. Since the Set operator is an SQL-only operator, it can help move the Analytic Function contained in the expression EXPR into the cursor. Furthermore, since the Helper Table is empty, the union all operation will have no logical effect. Most importantly, since Union All introduces almost no performance overhead, this workaround will minimize performance issues.

## **Restrictions**

There are significant restrictions for the above-mentioned solution. The expression operator currently (version 10.1.0.2 and earlier versions) does not allow the use of aggregation expressions (MIN, MAX, SUM, AVG etc.) because these expressions, when used outside the analytical functions context, typically require group by clause. A separate operator (the aggregator) is provided for these expressions. Since analytical functions heavily use aggregation expressions, it will not be possible to use the solution described above for any analytical function using aggregates. This will restrict the solution mostly to the ranking group of analytical functions.

### **Future direction**

Warehouse builder will increasingly support analytical functions. The next major release will eliminate the need for the workarounds described in this article by allowing the user to generate and deploy code only for the appropriate operating mode (set based for example), thus eliminating the need to introduce dummy operators in the mapping. The restriction regarding the use of aggregate expressions in the expression operator will be relaxed, thus making it possible for the user to enter the full range of analytic functions.

## **Conclusion**

This article illustrates how analytic functions can be used within Warehouse Builder. Similar results can be obtained by using other SQL-only operators such as dummy joins, aggregations etc. and paying attention not to alter the mapping logic. Care must also be taken to minimize the performance impact in this kind of workarounds.



White Paper Title

August 2004

Author: Igor Machin, Yu Gong

Contributing Authors: Óscar Javier Blanco Huerga

Oracle Corporation

World Headquarters

500 Oracle Parkway

Redwood Shores, CA 94065

U.S.A.

Worldwide Inquiries:

Phone: +1.650.506.7000

Fax: +1.650.506.7200

[www.oracle.com](http://www.oracle.com)

Oracle is a registered trademark of Oracle Corporation. Various product and service names referenced herein may be trademarks of Oracle Corporation. All other product and service names mentioned may be trademarks of their respective owners.

Copyright © 2001 Oracle Corporation

All rights reserved.