ROLLUP, CUBE, GROUPING Functions and GROUPING SETS

This article gives an overview of the functionality available for aggregation in data warehouses, focusing specifically on the information required for the Oracle Database SQL Expert (1Z0-047) (http://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=41&p_org_id=1001&lang=US&p_exam_id=1Z0_047) exam.

- Setup
- GROUP BY
- ROLLUP
- CUBE
- GROUPING Functions
  - GROUPING Function
  - GROUPING_ID Function
  - GROUP_ID Function
- GROUPING SETS
- Composite Columns
- Concatenated Groupings

For more information see:

- GROUP BY, ROLLUP and CUBE in Oracle ▶ (https://www.youtube.com/watch?v=CCm4IY-Ntfw)

Setup

The examples in this article will be run against the following simple dimension table.

Translate

```
DROP TABLE dimension_tab;
CREATE TABLE dimension_tab (
  fact_1_id   NUMBER NOT NULL,
  fact_2_id   NUMBER NOT NULL,
  fact_3_id   NUMBER NOT NULL,
  fact_4_id   NUMBER NOT NULL,
  sales_value NUMBER(10,2) NOT NULL
);

INSERT INTO dimension_tab
SELECT TRUNC(DBMS_RANDOM.value(low => 1, high => 3)) AS fact_1_id,
       TRUNC(DBMS_RANDOM.value(low => 1, high => 6)) AS fact_2_id,
       TRUNC(DBMS_RANDOM.value(low => 1, high => 11)) AS fact_3_id,
       TRUNC(DBMS_RANDOM.value(low => 1, high => 11)) AS fact_4_id,
       ROUND(DBMS_RANDOM.value(low => 1, high => 100), 2) AS sales_value
FROM   dual
CONNECT BY level <= 1000;
COMMIT;
```

To keep the queries and their output simple I am going to ignore the fact tables and also limit the number of distinct values in the columns of the dimension table.

GROUP BY

Let's start be reminding ourselves how the `GROUP BY` clause works. An aggregate function takes multiple rows of data returned by a query and aggregates them into a single result row.

```
SELECT SUM(sales_value) AS sales_value
FROM   dimension_tab;

SALES_VALUE
-----------
   50528.39

1 row selected.

SQL>
```

Including the `GROUP BY` clause limits the window of data processed by the aggregate function. This way we get an aggregated value for each distinct combination of values present in the columns listed in the `GROUP BY` clause. The number of rows we expect can be calculated by multiplying the number of distinct values of each column listed in the `GROUP BY` clause. In this case, if the rows were loaded randomly we would expect the number of distinct values for the first three columns in the table to be 2, 5 and 10 respectively. So using the `fact_1_id` column in the `GROUP BY` clause should give us 2 rows.

```
SELECT fact_1_id,
       COUNT(*) AS num_rows,
       SUM(sales_value) AS sales_value
FROM   dimension_tab
GROUP BY fact_1_id
ORDER BY fact_1_id;

 FACT_1_ID   NUM_ROWS SALES_VALUE
---------- ---------- -----------
         1        478    24291.35
         2        522    26237.04

2 rows selected.

SQL>
```

Including the first two columns in the `GROUP BY` clause should give us 10 rows (2*5), each with its aggregated values.

```
SELECT fact_1_id,
       fact_2_id,
       COUNT(*) AS num_rows,
       SUM(sales_value) AS sales_value
FROM   dimension_tab
GROUP BY fact_1_id, fact_2_id
ORDER BY fact_1_id, fact_2_id;

 FACT_1_ID  FACT_2_ID   NUM_ROWS SALES_VALUE
---------- ---------- ---------- -----------
         1          1         83     4363.55
         1          2         96     4794.76
         1          3         93     4718.25
         1          4        105     5387.45
         1          5        101     5027.34
         2          1        109     5652.84
         2          2         96     4583.02
         2          3        110     5555.77
         2          4        113     5936.67
         2          5         94     4508.74

10 rows selected.

SQL>
```

Including the first three columns in the `GROUP BY` clause should give us 100 rows (2*5*10).

```
SELECT fact_1_id,
       fact_2_id,
       fact_3_id,
       COUNT(*) AS num_rows,
       SUM(sales_value) AS sales_value
FROM   dimension_tab
GROUP BY fact_1_id, fact_2_id, fact_3_id
ORDER BY fact_1_id, fact_2_id, fact_3_id;

 FACT_1_ID  FACT_2_ID  FACT_3_ID   NUM_ROWS SALES_VALUE
---------- ---------- ---------- ---------- -----------
         1          1          1         10      381.61
         1          1          2          6      235.29
         1          1          3          7       270.7
         1          1          4         13      634.05
         1          1          5         10      602.36
         1          1          6          7      538.41
         1          1          7          5      245.87
         1          1          8          8      435.54
         1          1          9          8      506.59
         1          1         10          9      513.13
...
         2          5          1         14      714.84
         2          5          2         13      686.56
         2          5          3         13       579.5
         2          5          4         10      336.87
         2          5          5          5      215.17
         2          5          6          4      268.72
         2          5          7         14      667.22
         2          5          8          7      451.29
         2          5          9          8      365.24
         2          5         10          6      223.33

100 rows selected.

SQL>
```

ROLLUP

In addition to the regular aggregation results we expect from the `GROUP BY` clause, the `ROLLUP` extension produces group subtotals from right to left and a grand total. If "n" is the number of columns listed in the `ROLLUP`, there will be n+1 levels of subtotals.

```
SELECT fact_1_id,
       fact_2_id,
       SUM(sales_value) AS sales_value
FROM   dimension_tab
GROUP BY ROLLUP (fact_1_id, fact_2_id)
ORDER BY fact_1_id, fact_2_id;

 FACT_1_ID  FACT_2_ID SALES_VALUE
---------- ---------- -----------
         1          1     4363.55
         1          2     4794.76
         1          3     4718.25
         1          4     5387.45
         1          5     5027.34
         1               24291.35
         2          1     5652.84
         2          2     4583.02
         2          3     5555.77
         2          4     5936.67
         2          5     4508.74
         2               26237.04
                         50528.39

13 rows selected.

SQL>
```

Looking at the output in a SQL*Plus or a grid output, you can visually identify the rows containing subtotals as they have null values in the `ROLLUP` columns. It may be easier to spot when scanning down the output of the following query shown here (rollup.txt). Obviously, if the raw data contains null values, using this visual identification is not an accurate approach, but we will discuss this issue later.

```
SELECT fact_1_id,
       fact_2_id,
       fact_3_id,
       SUM(sales_value) AS sales_value
FROM   dimension_tab
GROUP BY ROLLUP (fact_1_id, fact_2_id, fact_3_id)
ORDER BY fact_1_id, fact_2_id, fact_3_id;
```

It is possible to do a partial rollup to reduce the number of subtotals calculated. The output from the following partial rollup is shown here (rollup-partial.txt).

```
SELECT fact_1_id,
       fact_2_id,
       fact_3_id,
       SUM(sales_value) AS sales_value
FROM   dimension_tab
GROUP BY fact_1_id, ROLLUP (fact_2_id, fact_3_id)
ORDER BY fact_1_id, fact_2_id, fact_3_id;
```

CUBE

In addition to the subtotals generated by the `ROLLUP` extension, the `CUBE` extension will generate subtotals for all combinations of the dimensions specified. If "n" is the number of columns listed in the `CUBE`, there will be 2^n subtotal combinations.

```
SELECT fact_1_id,
       fact_2_id,
       SUM(sales_value) AS sales_value
FROM   dimension_tab
GROUP BY CUBE (fact_1_id, fact_2_id)
ORDER BY fact_1_id, fact_2_id;

 FACT_1_ID  FACT_2_ID SALES_VALUE
---------- ---------- -----------
         1          1     4363.55
         1          2     4794.76
         1          3     4718.25
         1          4     5387.45
         1          5     5027.34
         1               24291.35
         2          1     5652.84
         2          2     4583.02
         2          3     5555.77
         2          4     5936.67
         2          5     4508.74
         2               26237.04
                    1    10016.39
                    2     9377.78
                    3    10274.02
                    4    11324.12
                    5     9536.08
                         50528.39

18 rows selected.

SQL>
```

As the number of dimensions increase, so do the combinations of subtotals that need to be calculated, as shown by the output of the following query, shown here (cube.txt).

```
SELECT fact_1_id,
       fact_2_id,
       fact_3_id,
       SUM(sales_value) AS sales_value
FROM   dimension_tab
GROUP BY CUBE (fact_1_id, fact_2_id, fact_3_id)
ORDER BY fact_1_id, fact_2_id, fact_3_id;
```

It is possible to do a partial cube to reduce the number of subtotals calculated. The output from the following partial cube is shown here (cube-partial.txt).

```
SELECT fact_1_id,
       fact_2_id,
       fact_3_id,
       SUM(sales_value) AS sales_value
FROM   dimension_tab
GROUP BY fact_1_id, CUBE (fact_2_id, fact_3_id)
ORDER BY fact_1_id, fact_2_id, fact_3_id;
```

GROUPING Functions

GROUPING

It can be quite easy to visually identify subtotals generated by rollups and cubes, but to do it programatically you really need something more accurate than the presence of null values in the grouping columns. This is where the `GROUPING` function comes in. It accepts a single column as a parameter and returns "1" if the column contains a null value generated as part of a subtotal by a `ROLLUP` or `CUBE` operation or "0" for any other value, including stored null values.

The following query is a repeat of a previous cube, but the `GROUPING` function has been added for each of the dimensions in the cube.

```
SELECT fact_1_id,
       fact_2_id,
       SUM(sales_value) AS sales_value,
       GROUPING(fact_1_id) AS f1g,
       GROUPING(fact_2_id) AS f2g
FROM   dimension_tab
GROUP BY CUBE (fact_1_id, fact_2_id)
ORDER BY fact_1_id, fact_2_id;

 FACT_1_ID  FACT_2_ID SALES_VALUE        F1G        F2G
---------- ---------- ----------- ---------- ----------
         1          1     4363.55          0          0
         1          2     4794.76          0          0
         1          3     4718.25          0          0
         1          4     5387.45          0          0
         1          5     5027.34          0          0
         1               24291.35          0          1
         2          1     5652.84          0          0
         2          2     4583.02          0          0
         2          3     5555.77          0          0
         2          4     5936.67          0          0
         2          5     4508.74          0          0
         2               26237.04          0          1
                    1    10016.39          1          0
                    2     9377.78          1          0
                    3    10274.02          1          0
                    4    11324.12          1          0
                    5     9536.08          1          0
                         50528.39          1          1

18 rows selected.

SQL>
```

From this we can see:

- F1G=0,F2G=0 : Represents a row containing regular subtotal we would expect from a `GROUP BY` operation.
- F1G=0,F2G=1 : Represents a row containing a subtotal for a distinct value of the `FACT_1_ID` column, as generated by `ROLLUP` and `CUBE` operations.
- F1G=1,F2G=0 : Represents a row containing a subtotal for a distinct value of the `FACT_2_ID` column, which we would only see in a `CUBE` operation.
- F1G=1,F2G=1 : Represents a row containing a grand total for the query, as generated by `ROLLUP` and `CUBE` operations.

It would now be easy to write a program to accurately process the data.

The `GROUPING` columns can used for ordering or filtering results.

```
SELECT fact_1_id,
       fact_2_id,
       SUM(sales_value) AS sales_value,
       GROUPING(fact_1_id) AS f1g,
       GROUPING(fact_2_id) AS f2g
FROM   dimension_tab
GROUP BY CUBE (fact_1_id, fact_2_id)
HAVING GROUPING(fact_1_id) = 1 OR GROUPING(fact_2_id) = 1
ORDER BY GROUPING(fact_1_id), GROUPING(fact_2_id);

 FACT_1_ID  FACT_2_ID SALES_VALUE        F1G        F2G
---------- ---------- ----------- ---------- ----------
         1               24291.35          0          1
         2               26237.04          0          1
                    4    11324.12          1          0
                    3    10274.02          1          0
                    2     9377.78          1          0
                    1    10016.39          1          0
                    5     9536.08          1          0
                         50528.39          1          1

8 rows selected.

SQL>
```

GROUPING_ID

The `GROUPING_ID` function provides an alternate and more compact way to identify subtotal rows. Passing the dimension columns as arguments, it returns a number indicating the `GROUP BY` level.

```
SELECT fact_1_id,
       fact_2_id,
       SUM(sales_value) AS sales_value,
       GROUPING_ID(fact_1_id, fact_2_id) AS grouping_id
FROM   dimension_tab
GROUP BY CUBE (fact_1_id, fact_2_id)
ORDER BY fact_1_id, fact_2_id;

 FACT_1_ID  FACT_2_ID SALES_VALUE GROUPING_ID
---------- ---------- ----------- -----------
         1          1     4363.55           0
         1          2     4794.76           0
         1          3     4718.25           0
         1          4     5387.45           0
         1          5     5027.34           0
         1               24291.35           1
         2          1     5652.84           0
         2          2     4583.02           0
         2          3     5555.77           0
         2          4     5936.67           0
         2          5     4508.74           0
         2               26237.04           1
                    1    10016.39           2
                    2     9377.78           2
                    3    10274.02           2
                    4    11324.12           2
                    5     9536.08           2
                         50528.39           3

18 rows selected.

SQL>
```

GROUP_ID

It's possible to write queries that return the duplicate subtotals, which can be a little confusing. The `GROUP_ID` function assigns the value "0" to the first set, and all subsequent sets get assigned a higher number. The following query forces duplicates to show the `GROUP_ID` function in action.

```
SELECT fact_1_id,
       fact_2_id,
       SUM(sales_value) AS sales_value,
       GROUPING_ID(fact_1_id, fact_2_id) AS grouping_id,
       GROUP_ID() AS group_id
FROM   dimension_tab
GROUP BY GROUPING SETS(fact_1_id, CUBE (fact_1_id, fact_2_id))
ORDER BY fact_1_id, fact_2_id;

 FACT_1_ID  FACT_2_ID SALES_VALUE GROUPING_ID   GROUP_ID
---------- ---------- ----------- ----------- ----------
         1          1     4363.55           0          0
         1          2     4794.76           0          0
         1          3     4718.25           0          0
         1          4     5387.45           0          0
         1          5     5027.34           0          0
         1               24291.35           1          1
         1               24291.35           1          0
         2          1     5652.84           0          0
         2          2     4583.02           0          0
         2          3     5555.77           0          0
         2          4     5936.67           0          0
         2          5     4508.74           0          0
         2               26237.04           1          1
         2               26237.04           1          0
                    1    10016.39           2          0
                    2     9377.78           2          0
                    3    10274.02           2          0
                    4    11324.12           2          0
                    5     9536.08           2          0
                         50528.39           3          0

20 rows selected.

SQL>
```

If necessary, you could then filter the results using the group.

```
SELECT fact_1_id,
       fact_2_id,
       SUM(sales_value) AS sales_value,
       GROUPING_ID(fact_1_id, fact_2_id) AS grouping_id,
       GROUP_ID() AS group_id
FROM   dimension_tab
GROUP BY GROUPING SETS(fact_1_id, CUBE (fact_1_id, fact_2_id))
HAVING GROUP_ID() = 0
ORDER BY fact_1_id, fact_2_id;

 FACT_1_ID  FACT_2_ID SALES_VALUE GROUPING_ID   GROUP_ID
---------- ---------- ----------- ----------- ----------
         1          1     4363.55           0          0
         1          2     4794.76           0          0
         1          3     4718.25           0          0
         1          4     5387.45           0          0
         1          5     5027.34           0          0
         1               24291.35           1          0
         2          1     5652.84           0          0
         2          2     4583.02           0          0
         2          3     5555.77           0          0
         2          4     5936.67           0          0
         2          5     4508.74           0          0
         2               26237.04           1          0
                    1    10016.39           2          0
                    2     9377.78           2          0
                    3    10274.02           2          0
                    4    11324.12           2          0
                    5     9536.08           2          0
                         50528.39           3          0

18 rows selected.

SQL>
```

GROUPING SETS

Calculating all possible subtotals in a cube, especially those with many dimensions, can be quite an intensive process. If you don't need all the subtotals, this can represent a considerable amount of wasted effort. The following cube with three dimensions gives 8 levels of subtotals (GROUPING_ID: 0-7), shown here (grouping-sets-cube.txt).

```
SELECT fact_1_id,
       fact_2_id,
       fact_3_id,
       SUM(sales_value) AS sales_value,
       GROUPING_ID(fact_1_id, fact_2_id, fact_3_id) AS grouping_id
FROM   dimension_tab
GROUP BY CUBE(fact_1_id, fact_2_id, fact_3_id)
ORDER BY fact_1_id, fact_2_id, fact_3_id;
```

If we only need a few of these levels of subtotaling we can use the `GROUPING SETS` expression and specify exactly which ones we need, saving us having to calculate the whole cube. In the following query we are only interested in subtotals for the " `FACT_1_ID, FACT_2_ID` " and " `FACT_1_ID, FACT_3_ID` " groups.

```
SELECT fact_1_id,
       fact_2_id,
       fact_3_id,
       SUM(sales_value) AS sales_value,
       GROUPING_ID(fact_1_id, fact_2_id, fact_3_id) AS grouping_id
FROM   dimension_tab
GROUP BY GROUPING SETS((fact_1_id, fact_2_id), (fact_1_id, fact_3_id))
ORDER BY fact_1_id, fact_2_id, fact_3_id;

 FACT_1_ID  FACT_2_ID  FACT_3_ID SALES_VALUE GROUPING_ID
---------- ---------- ---------- ----------- -----------
         1          1                4363.55           1
         1          2                4794.76           1
         1          3                4718.25           1
         1          4                5387.45           1
         1          5                5027.34           1
         1                     1      2737.4           2
         1                     2     1854.29           2
         1                     3     2090.96           2
         1                     4     2605.17           2
         1                     5     2590.93           2
         1                     6      2506.9           2
         1                     7     1839.85           2
         1                     8     2953.04           2
         1                     9     2778.75           2
         1                    10     2334.06           2
         2          1                5652.84           1
         2          2                4583.02           1
         2          3                5555.77           1
         2          4                5936.67           1
         2          5                4508.74           1
         2                    1     3512.69          2
         2                    2     2847.94          2
         2                    3      2972.5          2
         2                    4     2534.06          2
         2                    5     3115.99          2
         2                    6     2775.85          2
         2                    7     2208.19          2
         2                    8     2358.55          2
         2                    9     1884.11          2
         2                   10     2027.16          2

30 rows selected.

SQL>
```

Notice how we have gone from returning 198 rows with 8 subtotal levels in the cube, to just 30 rows with 2 subtotal levels.

Composite Columns

`ROLLUP` and `CUBE` consider each column independently when deciding which subtotals must be calculated. For `ROLLUP` this means stepping back through the list to determine the groupings.

```
ROLLUP (a, b, c)
(a, b, c)
(a, b)
(a)
()
```

`CUBE` creates a grouping for every possible combination of columns.

```
CUBE (a, b, c)
(a, b, c)
(a, b)
(a, c)
(a)
(b, c)
(b)
(c)
()
```

Composite columns allow columns to be grouped together with braces so they are treated as a single unit when determining the necessary groupings. In the following `ROLLUP` columns "a" and "b" have been turned into a composite column by the additional braces. As a result the group of "a" is not longer calculated as the column "a" is only present as part of the composite column in the statement.

```
ROLLUP ((a, b), c)
(a, b, c)
(a, b)
()

Not considered:
(a)
```

In a similar way, the possible combinations of the following `CUBE` are reduced because references to "a" or "b" individually are not considered as they are treated as a single column when the groupings are determined.

```
CUBE ((a, b), c)
(a, b, c)
(a, b)
(c)
()

Not considered:
(a, c)
(a)
(b, c)
(b)
```

The impact of this is shown clearly in the follow two statements, whose output is shown here (composite-column-1.txt) and here (composite-column-2.txt). The regular cube returns 198 rows and 8 groups (0-7), while the cube with the composite column returns only 121 rows with 4 groups (0, 1, 6, 7)

```
-- Regular Cube.
SELECT fact_1_id,
       fact_2_id,
       fact_3_id,
       SUM(sales_value) AS sales_value,
       GROUPING_ID(fact_1_id, fact_2_id, fact_3_id) AS grouping_id
FROM   dimension_tab
GROUP BY CUBE(fact_1_id, fact_2_id, fact_3_id)
ORDER BY fact_1_id, fact_2_id, fact_3_id;

-- Cube with composite column.
SELECT fact_1_id,
       fact_2_id,
       fact_3_id,
       SUM(sales_value) AS sales_value,
       GROUPING_ID(fact_1_id, fact_2_id, fact_3_id) AS grouping_id
FROM   dimension_tab
GROUP BY CUBE((fact_1_id, fact_2_id), fact_3_id)
ORDER BY fact_1_id, fact_2_id, fact_3_id;
```

Concatenated Groupings

Concatenated groupings are defined by putting together multiple `GROUPING SETS`, `CUBE`s or `ROLLUP`s separated by commas. The resulting groupings are the cross-product of all the groups produced by the individual grouping sets. It might be a little easier to understand what this means by looking at an example. The following `GROUPING SET` results in 2 groups of subtotals, one for the `fact_1_id` column and one for the `fact_id_2` column.

```
SELECT fact_1_id,
       fact_2_id,
       SUM(sales_value) AS sales_value,
       GROUPING_ID(fact_1_id, fact_2_id) AS grouping_id
FROM   dimension_tab
GROUP BY GROUPING SETS(fact_1_id, fact_2_id)
ORDER BY fact_1_id, fact_2_id;

 FACT_1_ID  FACT_2_ID SALES_VALUE GROUPING_ID
---------- ---------- ----------- -----------
         1               24291.35           1
         2               26237.04           1
                    1    10016.39           2
                    2     9377.78           2
                    3    10274.02           2
                    4    11324.12           2
                    5     9536.08           2

7 rows selected.

SQL>
```

The next `GROUPING SET` results in another 2 groups of subtotals, one for the `fact_3_id` column and one for the `fact_4_id` column.

```
SELECT fact_3_id,
       fact_4_id,
       SUM(sales_value) AS sales_value,
       GROUPING_ID(fact_3_id, fact_4_id) AS grouping_id
FROM   dimension_tab
GROUP BY GROUPING SETS(fact_3_id, fact_4_id)
ORDER BY fact_3_id, fact_4_id;

 FACT_3_ID  FACT_4_ID SALES_VALUE GROUPING_ID
---------- ---------- ----------- -----------
         1                6250.09           1
         2                4702.23           1
         3                5063.46           1
         4                5139.23           1
         5                5706.92           1
         6                5282.75           1
         7                4048.04           1
         8                5311.59           1
         9                4662.86           1
        10                4361.22           1
                    1     4718.55           2
                    2      5439.1           2
                    3      4643.4           2
                    4      4515.3           2
                    5     5110.27           2
                    6     5910.78           2
                    7     4987.22           2
                    8     4846.25           2
                    9     5458.82           2
                   10      4898.7           2

20 rows selected.

SQL>
```

If we combine them together into a concatenated grouping we get 4 groups of subtotals. The output of the following query is shown here (concatenated-groupings.txt).

```
SELECT fact_1_id,
       fact_2_id,
       fact_3_id,
       fact_4_id,
       SUM(sales_value) AS sales_value,
       GROUPING_ID(fact_1_id, fact_2_id, fact_3_id, fact_4_id) AS grouping_
id
FROM   dimension_tab
GROUP BY GROUPING SETS(fact_1_id, fact_2_id), GROUPING SETS(fact_3_id, fact
_4_id)
ORDER BY fact_1_id, fact_2_id, fact_3_id, fact_4_id;
```

The output from the previous three queries produce the following groupings.

```
GROUPING SETS(fact_1_id, fact_2_id)
(fact_1_id)
(fact_2_id)

GROUPING SETS(fact_3_id, fact_4_id)
(fact_3_id)
(fact_4_id)

GROUPING SETS(fact_1_id, fact_2_id), GROUPING SETS(fact_3_id, fact_4_id)
(fact_1_id, fact_3_id)
(fact_1_id, fact_4_id)
(fact_2_id, fact_3_id)
(fact_2_id, fact_4_id)
```

So we can see the final cross-product of the two `GROUPING SETS` that make up the concatenated grouping. A generic summary would be as follows.

```
GROUPING SETS(a, b), GROUPING SETS(c, d)
(a, c)
(a, d)
(b, c)
(b, d)
```

For more information see:

- GROUP BY, ROLLUP and CUBE in Oracle (https://www.youtube.com/watch?v=CCm4IY-Ntfw)
- SQL for Aggregation in Data Warehouses (http://docs.oracle.com/cd/E11882_01/server.112/e16579/aggreg.htm)

Hope this helps. Regards Tim...

Back to the Top.

17 comments, read/add them... (/misc/comments?page_id=940)

Home (/) | Articles (/articles/articles) | Scripts (/dba/scripts) | Blog (/blog/) | Certification (/misc/ocp-certification) | Misc (/misc/miscellaneous) | About (/misc/site-info)

About Tim Hall (/misc/site-info#biog)
Copyright & Disclaimer (/misc/site-info#copyright)