Using Partitioned Outer Join to Fill Gaps in Sparse Data

Purpose

In this module you learn how to use the new SQL Join syntax in the Oracle Database 10 *g* SQL to fill gaps in sparse data.

Topics

This module will discuss the following:

Overview

Back to List of Topics

Oracle Database 10 *g* Partitioned Outer Join Clause Overview

Data is normally stored in sparse form. That is, if no value exists for a given time, no row exists in the fact table. However, time series calculations can be performed most easily when data is dense along the time dimension. This is because dense data will fill a consistent number of rows for each period, which in turn makes it simple to use the analytic windowing functions with physical offsets. Refer to *Chapter 21: Data Warehousing Guide* for more information.

To overcome the problem of sparsity, you can use a partitioned outer join to fill the gaps in a time series. Such a join extends the conventional outer join syntax by applying the outer join to each logical partition defined in a query. The Oracle database logically partitions the rows in your query based on the expression you specify in the `PARTITION BY` clause. The result of a partitioned outer join is a `UNION` of the outer joins of each of the groups in the logically partitioned table with the table on the other side of the join.

Note that you can use this type of join to fill the gaps in any dimension, not just the time dimension. In this module, you will focus on the time dimension because it is the dimension most frequently used as a basis for comparisons.

Prerequisites

Back to List

Before starting this module, you should have performed the following:

☐ **1.** Completed the Configuring Linux for the Installation of Oracle Database 10g lesson

☐ **2.** Completed the Installing the Oracle Database 10g on Linux lesson

☐ **3.** Download the outer_j.zip into your working directory.

Syntax

Back to List of Topics

The syntax for partitioned outer join extends the ANSI SQL `JOIN` clause with the phrase `PARTITION BY` followed by an expression list. The expressions in the list specify the group to which the outer join is applied. The following are the two forms of syntax normally used for partitioned outer join:

```
SELECT
select_expression
FROM
table_reference
  PARTITION BY (
expr
 [,
expr
 ]... )
  RIGHT OUTER JOIN
table_reference
```

```
SELECT
select_expression
FROM
table_reference
  LEFT OUTER JOIN
table_reference
  PARTITION BY {
expr
 [,
expr
 ]...)
```

Note that `FULL OUTER JOIN` is not supported with a partitioned outer join. Refer to the Oracle Database 10 *g* SQL Reference for further information regarding syntax and restrictions.

Sample of Sparse Data

Back to List of Topics

A typical situation with a sparse dimension is shown in the following example, which computes the weekly sales and year-to-date sales for the product 'Bounce' for weeks 20-30 in 2000 and 2001:

```
SELECT

  SUBSTR(p.Prod_Name,1,15) Product_Name,

  t.Calendar_Year Year,

  t.Calendar_Week_Number Week,

  SUM(Amount_Sold) Sales

FROM Sales s, Times t, Products p

WHERE s.Time_id = t.Time_id AND

  s.Prod_id = p.Prod_id AND

  p.Prod_name IN ('Bounce') AND

  t.Calendar_Year IN (2000,2001)  AND

  t.Calendar_Week_Number BETWEEN  20 AND 30

GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number;
```

```
PRODUCT_NAME          YEAR       WEEK      SALES
--------------- ---------- ---------- ----------
Bounce                2000         20        801
Bounce                2000         21    4062.24
Bounce                2000         22    2043.16
Bounce                2000         23    2731.14
Bounce                2000         24    4419.36
Bounce                2000         27    2297.29
Bounce                2000         28    1443.13
Bounce                2000         29    1927.38
Bounce                2000         30    1927.38
Bounce                2001         20     1483.3
Bounce                2001         21    4184.49
Bounce                2001         22    2609.19
Bounce                2001         23    1416.95
Bounce                2001         24    3149.62
Bounce                2001         25    2645.98
Bounce                2001         27    2125.12
Bounce                2001         29    2467.92
Bounce                2001         30    2620.17

18 rows selected.
```

In this example you would expect 22 rows of data (11 weeks each from 2 years) if the data were dense. However you see only 18 rows because weeks 25 and 26 are missing in 2000, and weeks 26 and 28 are missing in 2001.

Filling Gaps in Data

Back to List of Topics

Gaps in time series make calculations such as year-over-year comparisons hard to compute. When there are no gaps, you can compare data by referring from one row to another row a fixed distance away using the analytic functions `LEAD()` and `LAG()`. For instance, if you retrieve month level data and would like refer to data from 12 months ago, it is convenient to access data 12 rows before the current value. You cannot reliably use the `LEAD()` and `LAG()` functions when the number of rows per period (or whatever other dimension used as the divider) is inconsistent.

How can you fill in the gaps in the preceding example with a partitioned outer join?

You can take the sparse data of our query above and do a partitioned outer join with a dense set of time data. In the query shown below, the original query is aliased as "v" and the data retrieved from the times table is aliased as "t". Here you see 22 rows because there are no gaps in the series. The four added rows each have 0 as their Sales value set to 0 by using the `NVL()` function..

```
SELECT Product_Name, t.Year, t.Week, NVL(Sales,0) dense_sales

FROM

 (SELECT

   SUBSTR(p.Prod_Name,1,15) Product_Name,

   t.Calendar_Year Year,

   t.Calendar_Week_Number Week,

   SUM(Amount_Sold) Sales

   FROM Sales s, Times t, Products p

   WHERE s.Time_id = t.Time_id AND

     s.Prod_id = p.Prod_id AND

     p.Prod_name IN ('Bounce') AND

     t.Calendar_Year IN (2000,2001)  AND

     t.Calendar_Week_Number BETWEEN  20 AND 30

   GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number

  )  v


 PARTITION BY  (v.Product_Name)

   RIGHT OUTER JOIN

     (SELECT DISTINCT

       Calendar_Week_Number Week,
        Calendar_Year Year
       FROM Times
       WHERE Calendar_Year in (2000, 2001)
         AND Calendar_Week_Number BETWEEN 20 AND 30
     ) t
  ON (v.week = t.week AND v.Year = t.Year)
ORDER BY t.year, t.week;
```

```
PRODUCT_NAME          YEAR       WEEK DENSE_SALES
--------------- ---------- ---------- -----------
Bounce                2000         20         801
Bounce                2000         21     4062.24
Bounce                2000         22     2043.16
Bounce                2000         23     2731.14
Bounce                2000         24     4419.36
Bounce                2000         25           0
Bounce                2000         26           0
Bounce                2000         27     2297.29
Bounce                2000         28     1443.13
Bounce                2000         29     1927.38
Bounce                2000         30     1927.38
Bounce                2001         20      1483.3
Bounce                2001         21     4184.49
Bounce                2001         22     2609.19
Bounce                2001         23     1416.95
Bounce                2001         24     3149.62
Bounce                2001         25     2645.98
Bounce          2001      26        0

Bounce          2001      27    2125.12

Bounce          2001      28        0

Bounce          2001      29    2467.92

Bounce          2001      30    2620.17

22 rows selected.
```

Note that in the query above a `WHERE` condition for weeks between 20 and 30 is placed in the inline view for the time dimension. This step reduces the number of rows handled by the outer join, saving processing time.

Filling Gaps in Data and Using Analytic SQL Functions

Back to List of Topics

How do you combine this technique with analytic SQL functions to get cumulative sales for the desired weeks?

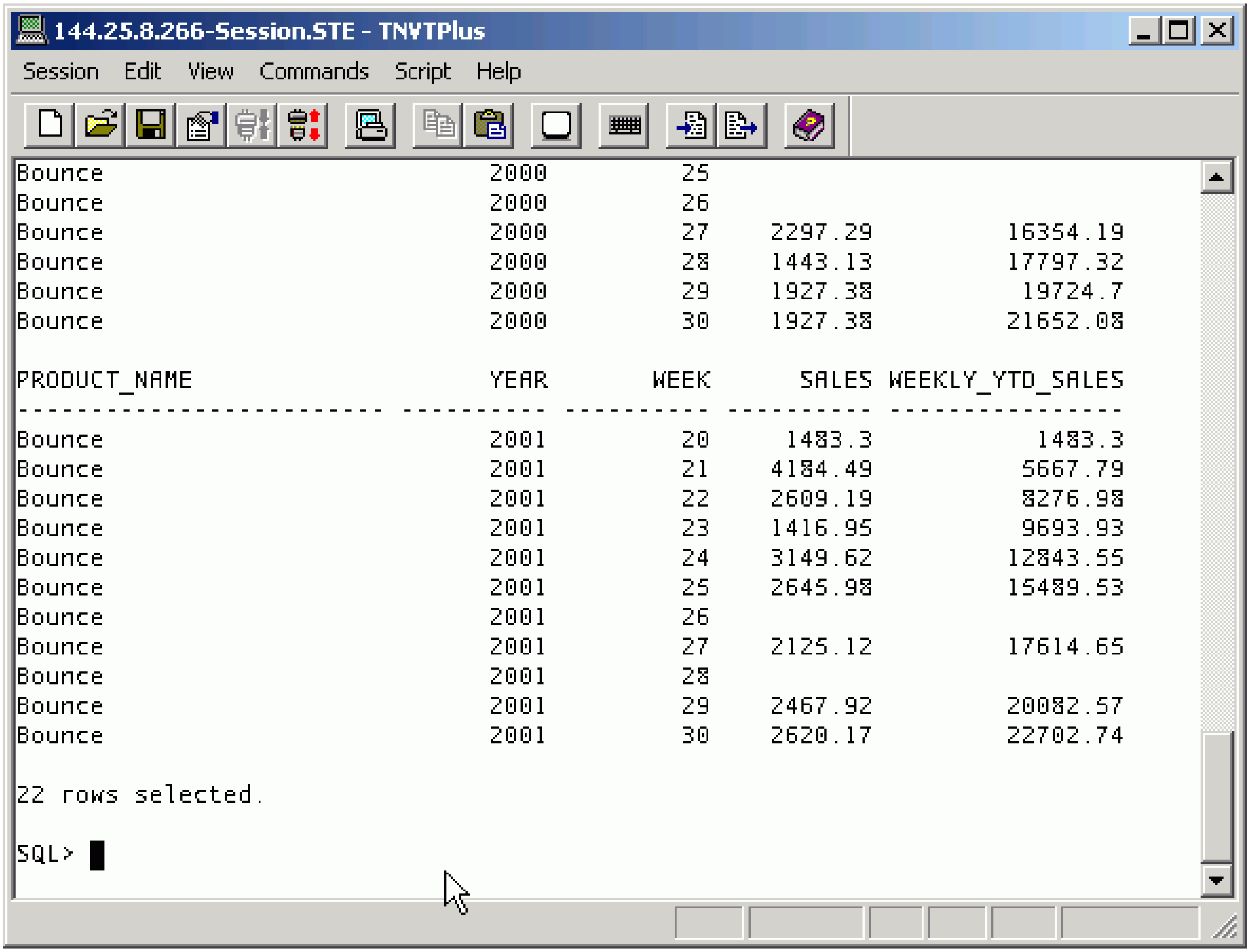
☐ **1.** From a terminal window, execute the following command(s):

```
cd wkdir
sqlplus sh/sh@orcl
@fg
```

The **`fg.sql`** script contains the following:

```
SELECT Product_Name, t.Year, t.Week, Sales, Weekly_ytd_sales
 FROM
  (SELECT
    SUBSTR(p.Prod_Name,1,15) Product_Name,
    t.Calendar_Year Year,
    t.Calendar_Week_Number Week,
    NVL(SUM(Amount_Sold),0) Sales,
    SUM(SUM(Amount_Sold)) OVER
      (PARTITION BY p.Prod_Name, t.Calendar_Year
      ORDER BY t.Calendar_Week_Number) Weekly_ytd_sales
    FROM Sales s, Times t, Products p
    WHERE s.Time_id = t.Time_id AND
      s.Prod_id = p.Prod_id AND
      p.Prod_name IN ('Bounce') AND
      t.Calendar_Year IN (2000,2001)  AND
```

```
       t.Calendar_Week_Number BETWEEN  20 AND 30
     GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number
    )  v
   PARTITION BY  (v.Product_Name)
   RIGHT OUTER JOIN
    (SELECT DISTINCT
       Calendar_Week_Number Week,
       Calendar_Year Year
     FROM Times
       WHERE Calendar_Year in (2000, 2001)
    ) t
    ON (v.week = t.week AND v.Year = t.Year)
 WHERE t.Week BETWEEN 20 AND 30
 ORDER BY 1, 2, 3;
```



```
Bounce                          2000         25
Bounce                          2000         26
Bounce                          2000         27    2297.29         16354.19
Bounce                          2000         28    1443.13         17797.32
Bounce                          2000         29    1927.38          19724.7
Bounce                          2000         30    1927.38         21652.08

PRODUCT_NAME                    YEAR       WEEK      SALES WEEKLY_YTD_SALES
------------------------- ---------- ---------- ---------- ----------------
Bounce                          2001         20     1483.3           1483.3
Bounce                          2001         21    4184.49          5667.79
Bounce                          2001         22    2609.19          8276.98
Bounce                          2001         23    1416.95          9693.93
Bounce                          2001         24    3149.62         12843.55
Bounce                          2001         25    2645.98         15489.53
Bounce                          2001         26
Bounce                          2001         27    2125.12         17614.65
Bounce                          2001         28
Bounce                          2001         29    2467.92         20082.57
Bounce                          2001         30    2620.17         22702.74

22 rows selected.

SQL>
```

In this query, the weekly year-to-date sales are calculated alongside the weekly sales. The NULL values that the partitioned outer join inserts in making the time series dense are handled in the usual way: the `SUM` function treats them as 0's.

Replacing NULLs with the Nearest Non-NULL Value

Back to List of Topics

There are queries in which a partitioned outer join will return rows with NULL values, but you may want those rows to hold the most recent non-NULL value in the series. That is, if you want to have NULLs replaced with the first non-NULL value you see as you scan upward in a column.

Inventory tables, which track quantity of units available for various products, are a common case needing such output. Inventory tables are sparse: like sales tables, they need only store a row for a product when there is an event. For a sales table the event is a sale, and for the inventory table, the event is a change in quantity available for a product. If you make the inventory's time dimension dense, you want to see a quantity value for each day. The value to output is the most recent non-NULL value. Note that this differs from the prior example with cumulative sales. In that query, the cumulative sum calculation treats NULLs as 0s, so it presents correct values. That approach cannot work with inventory and similar tables, since the value to place in rows with NULLs is not a sum.

Here an example is presented of partitioned outer join with an inventory table. It replaces NULLs with the nearest non-NULL value.

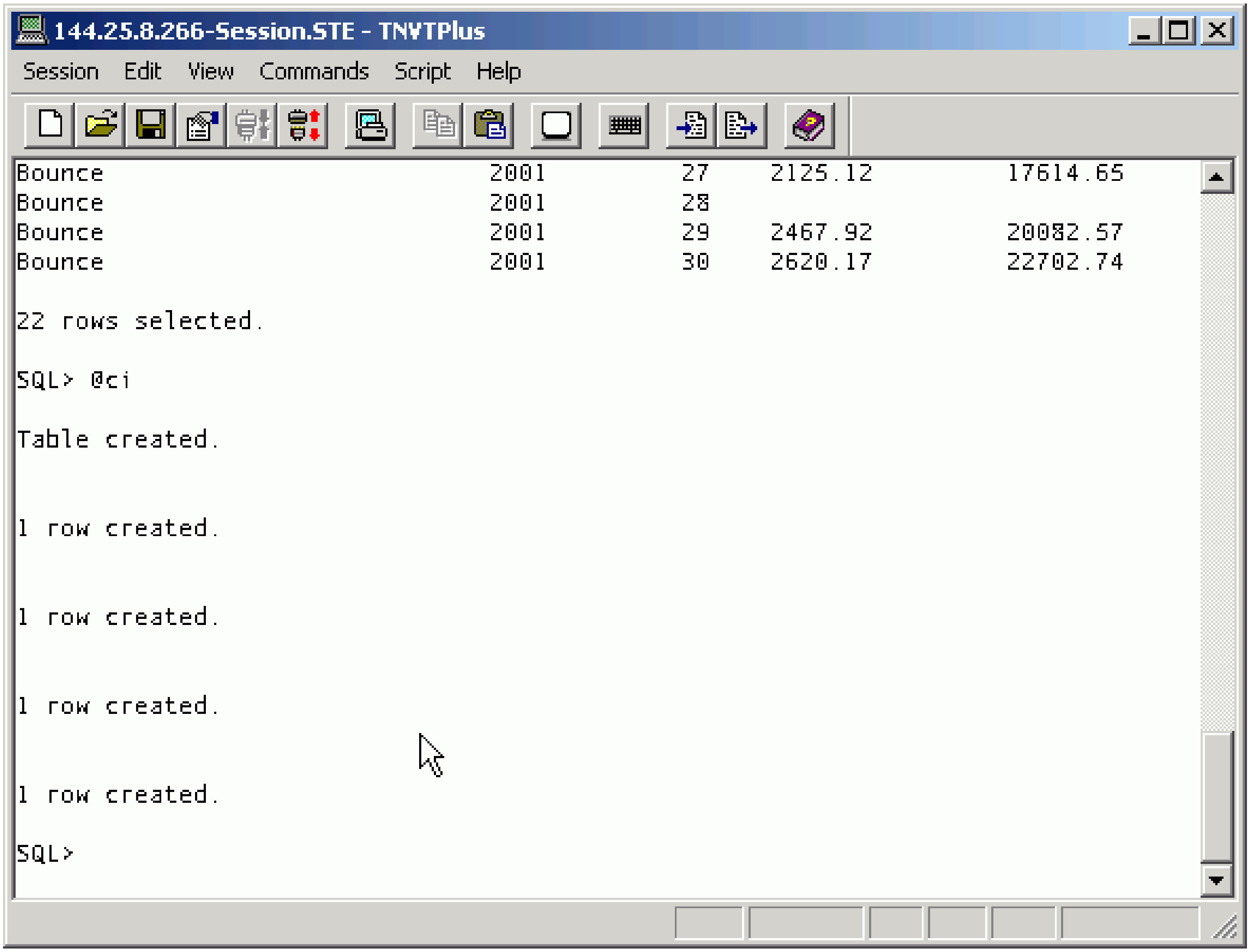
First, you create a small inventory table with two products, each product having entries for two days. The "bottle" product has 10 units in stock on April 1, and the "can" product has 15 units in stock on April 1.

☐ **1.** First you will create a small inventory table with two products, each product having entries for two days. The "bottle" product has 10 units in stock on April 1, and the "can" product has 15 units in stock on April 1. Execute the following SQL*Plus script:

```
@ci
```

The **`ci.sql`** script contains the following:

```
CREATE TABLE inventory (
         time_id DATE,
         product VARCHAR2(10),
         quant NUMBER);
INSERT INTO inventory VALUES
         (TO_DATE('01/04/01', 'DD/MM/YY'), 'bottle', 10);
INSERT INTO inventory VALUES
         (TO_DATE('06/04/01', 'DD/MM/YY'), 'bottle', 8);
INSERT INTO inventory VALUES
         (TO_DATE('01/04/01', 'DD/MM/YY'), 'can', 15);
INSERT INTO inventory VALUES
         (TO_DATE('04/04/01', 'DD/MM/YY'), 'can', 11);
```


144.25.8.266-Session.STE - TNVTPlus
Session Edit View Commands Script Help

```
Bounce                         2001        27     2125.12         17614.65
Bounce                         2001        28
Bounce                         2001        29     2467.92         20082.57
Bounce                         2001        30     2620.17         22702.74

22 rows selected.

SQL> @ci

Table created.


1 row created.


1 row created.


1 row created.


1 row created.

SQL>
```

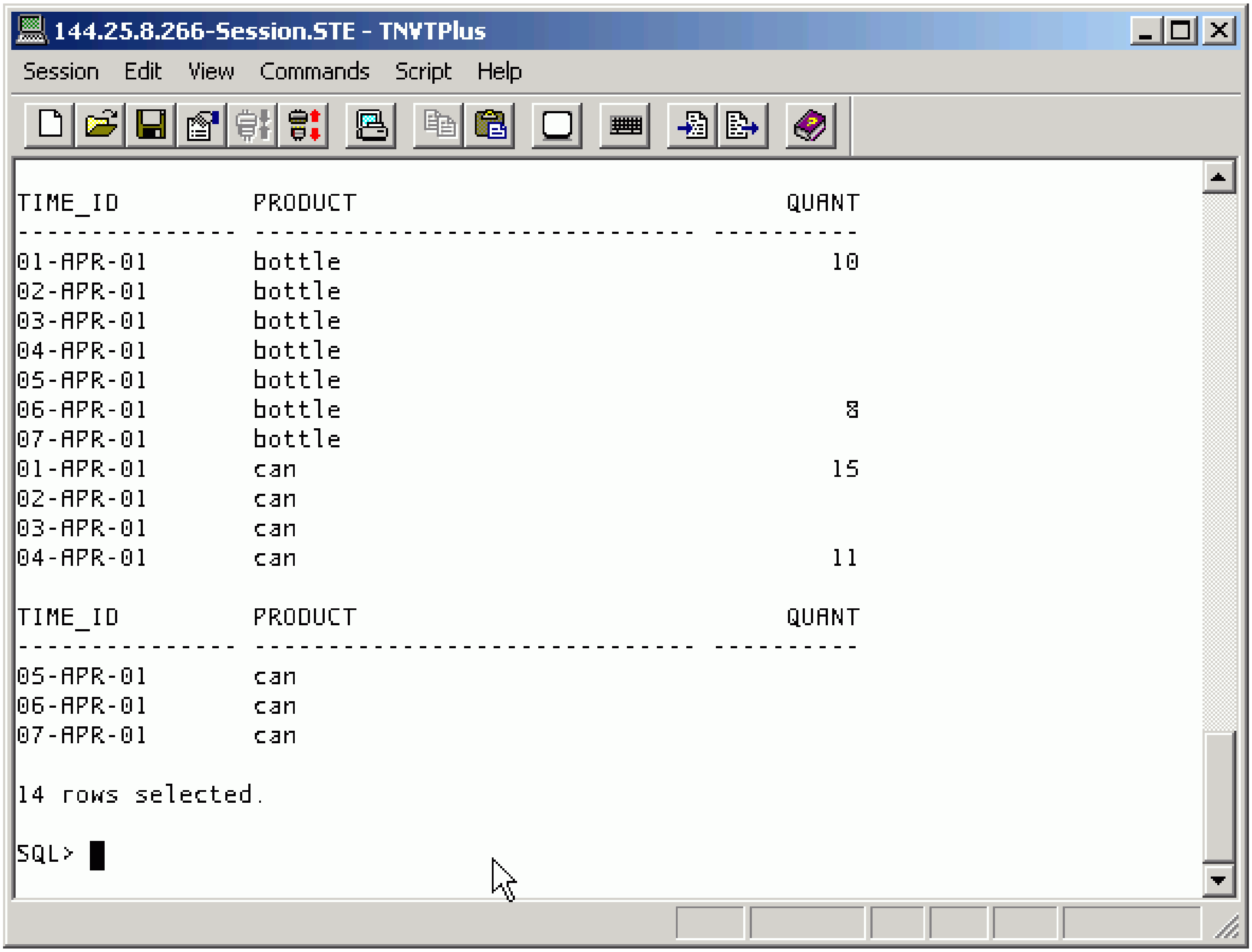
☐ **2.** Now you will use a partitioned outer join to to see the quantity available for each product on each day of the range April 1 through April 7. If you use a partitioned outer join to query this table without considering the rows with NULL values, the results are misleading. Execute the following SQL*Plus script:

```
@nn
```

The **`nn.sql`** script contains the following:

```
SELECT times.time_id, product, quant
FROM inventory

PARTITION BY (product)
  RIGHT OUTER JOIN times
  ON (times.time_id = inventory.time_id)
```

```
WHERE times.time_id BETWEEN TO_DATE('01/04/01', 'DD/MM/YY')
  AND TO_DATE('07/04/01', 'DD/MM/YY')
ORDER BY  2,1;
```


144.25.8.266-Session.STE - TNVTPlus
Session Edit View Commands Script Help

```
TIME_ID         PRODUCT                              QUANT
--------------- ------------------------------ ----------
01-APR-01       bottle                                 10
02-APR-01       bottle
03-APR-01       bottle
04-APR-01       bottle
05-APR-01       bottle
06-APR-01       bottle                                  8
07-APR-01       bottle
01-APR-01       can                                    15
02-APR-01       can
03-APR-01       can
04-APR-01       can                                    11

TIME_ID         PRODUCT                              QUANT
--------------- ------------------------------ ----------
05-APR-01       can
06-APR-01       can
07-APR-01       can

14 rows selected.

SQL>
```

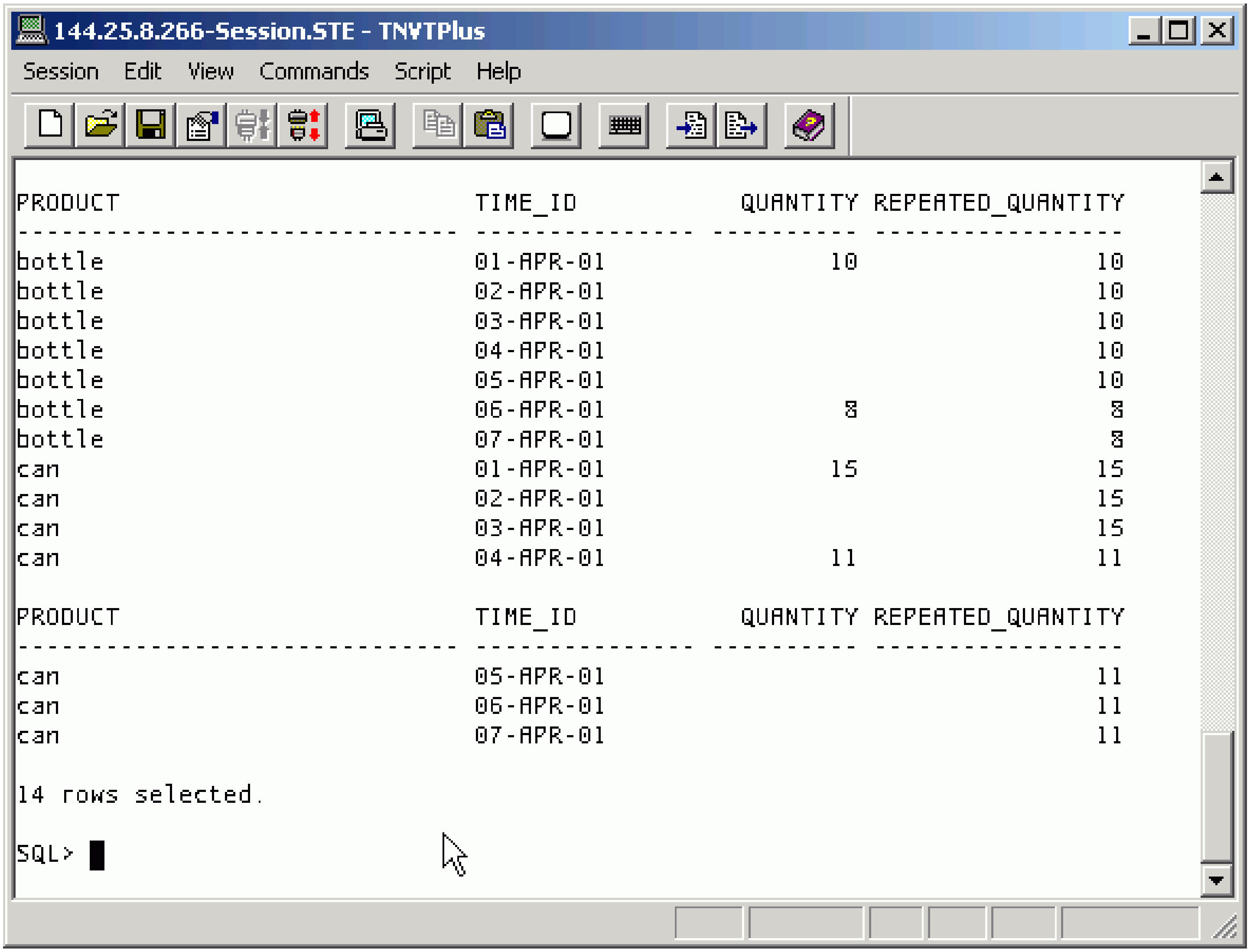
The results above are not what you wanted: you know that the quantities available for bottle and can in the NULL-value rows were simply the most recent non_NULL value. For instance, on April 2-5 for bottle, you want to see the quantity 10.

☐ **3.** To show the desired results, you want to take advantage of a new keyword added to the FIRST_VALUE and LAST_VALUE functions in Oracle Database 10g. You can specify IGNORE NULLS in the argument list of either of these functions, and they will return the closest non-NULL value. Execute the following SQL*Plus script:

```
@nn2
```

The **`nn2.sql`** script contains the following:

```
WITH v1 AS
(SELECT time_id
 FROM times
 WHERE times.time_id BETWEEN
  TO_DATE('01/04/01', 'DD/MM/YY')
  AND TO_DATE('07/04/01', 'DD/MM/YY'))
 SELECT product, time_id, quant quantity,
     LAST_VALUE(quant IGNORE NULLS)
  OVER (PARTITION BY product ORDER BY time_id)
    repeated_quantity
 FROM
  (SELECT product, v1.time_id, quant
   FROM inventory PARTITION BY (product)
     RIGHT OUTER JOIN v1
     ON (v1.time_id = inventory.time_id))
ORDER BY 1, 2;
```



```
PRODUCT                        TIME_ID          QUANTITY REPEATED_QUANTITY
------------------------------ --------------- ---------- -----------------
bottle                         01-APR-01               10                10
bottle                         02-APR-01                                 10
bottle                         03-APR-01                                 10
bottle                         04-APR-01                                 10
bottle                         05-APR-01                                 10
bottle                         06-APR-01                8                 8
bottle                         07-APR-01                                  8
can                            01-APR-01               15                15
can                            02-APR-01                                 15
can                            03-APR-01                                 15
can                            04-APR-01               11                11

PRODUCT                        TIME_ID          QUANTITY REPEATED_QUANTITY
------------------------------ --------------- ---------- -----------------
can                            05-APR-01                                 11
can                            06-APR-01                                 11
can                            07-APR-01                                 11

14 rows selected.

SQL>
```

Period-to-Period Comparison of One Time Level

Back to List of Topics

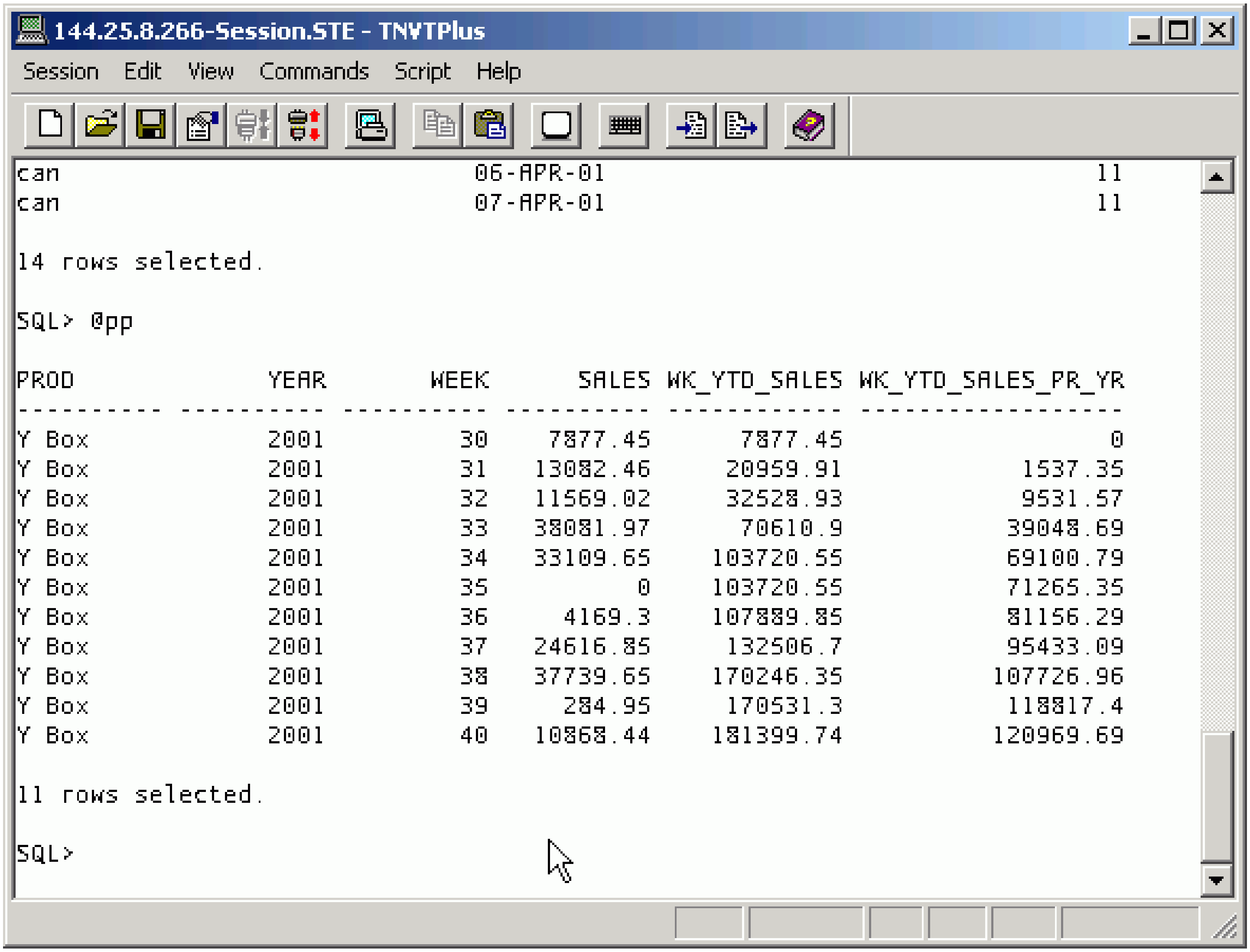
In the next task, you will use the outer join feature to compare values across time periods. Specifically, you will calculate a year-over-year sales comparison at the week level. The query will return on the same row, for each product, the year-to-date sales for each week of 2001 with that of 2000.

☐ **1.** To improve readability of the query and focus on the partitioned outer join, use a `WITH` clause to start the query. Execute the following SQL*Plus script:

```
@pp
```

The **`pp.sql`** script contains the following:

```
WITH v AS
  (SELECT
     p.Prod_Name Product_Name,
     t.Calendar_Year Year,
     t.Calendar_Week_Number Week,
     SUM(Amount_Sold) Sales
   FROM Sales s, Times t, Products p
   WHERE s.Time_id = t.Time_id AND
         s.Prod_id = p.Prod_id AND
         p.Prod_name in ('Y Box') AND
         t.Calendar_Year in (2000,2001) AND
         t.Calendar_Week_Number BETWEEN 30 AND 40
   GROUP BY p.Prod_Name, t.Calendar_Year, t.Calendar_Week_Number
  )
SELECT substr(Product_Name,1,12) Prod,
  Year,
  Week,
  Sales,
  Weekly_ytd_sales,
  Weekly_ytd_sales_prior_year
FROM
  (SELECT --Start of year_over_year sales
     Product_Name, Year, Week, Sales, Weekly_ytd_sales,
     LAG(Weekly_ytd_sales, 1) OVER
    (PARTITION BY Product_Name, Week ORDER BY Year)
     Weekly_ytd_sales_prior_year
   FROM
     (SELECT --
Start of dense_sales
        v.Product_Name Product_Name,
        t.Year Year,
        t.Week Week,
        NVL(v.Sales,0) Sales,
        SUM(NVL(v.Sales,0)) OVER

(PARTITION BY v.Product_Name, t.Year
        ORDER BY t.week) weekly_ytd_sales
      FROM v
        PARTITION BY (v.Product_Name)
      RIGHT OUTER JOIN
       (SELECT DISTINCT
          Calendar_Week_Number Week,
          Calendar_Year Year
        FROM Times
        WHERE Calendar_Year IN (2000, 2001)
       ) t
   ON (v.week = t.week AND v.Year = t.Year)
      )
dense_sales
      )
year_over_year_sales
WHERE Year = 2001 AND
      Week BETWEEN 30 AND 40
ORDER BY 1, 2, 3;
```



```
can                            06-APR-01                                 11
can                            07-APR-01                                 11

14 rows selected.

SQL> @pp

PROD             YEAR       WEEK      SALES WK_YTD_SALES WK_YTD_SALES_PR_YR
---------- ---------- ---------- ---------- ------------ ------------------
Y Box            2001         30    7877.45      7877.45                  0
Y Box            2001         31   13082.46     20959.91            1537.35
Y Box            2001         32   11569.02     32528.93            9531.57
Y Box            2001         33   38081.97      70610.9           39048.69
Y Box            2001         34   33109.65    103720.55           69100.79
Y Box            2001         35          0    103720.55           71265.35
Y Box            2001         36     4169.3    107889.85           81156.29
Y Box            2001         37   24616.85     132506.7           95433.09
Y Box            2001         38   37739.65    170246.35          107726.96
Y Box            2001         39     284.95     170531.3           118817.4
Y Box            2001         40   10868.44    181399.74          120969.69

11 rows selected.

SQL>
```

In the FROM clause of the in-line view DENSE_SALES , a partitioned outer join of aggregate view v and time view t is used to fill gaps in the sales data along the time dimension. The output of the partitioned outer join is then processed by the analytic function SUM ... OVER to compute the weekly year-to-date sales (the "weekly_ytd_sales" column). Thus, the view DENSE_SALES computes the year-to-date sales data for each week, including those missing in the aggregate view s .

The in-line view YEAR_OVER_YEAR_SALES then computes the year ago weekly year-to-date sales using the

`LAG` function. The `LAG` function labeled " `weekly_ytd_sales_prior_year` " specifies a `PARTITION BY` clause that pairs rows for the same week of years 2000 and 2001 into a single partition. An offset of 1 is passed to the `LAG` function to get the weekly year to date sales for the prior year.

The outermost query block selects data from `YEAR_OVER_YEAR_SALES` with the condition `yr = 2001` , and thus the query returns, for each product, its weekly year-to-date sales in the specified weeks of years 2001 and 2000.

Example of Period-to-Period Comparison for Multiple Time Levels

Back to List of Topics

While the prior example showed you a way to create comparisons for a single time level, it is even more useful to handle multiple time levels in a single query. For instance, you can compare sales versus the prior period at the year, quarter, month and day levels.

For the next task, you will create a query which performs a year-over-year comparison of year-to-date sales for all levels of our time hierarchy.

Several steps are needed to perform this task. The goal is a single query with comparisons at the day, week, month, quarter, and year level. You will use a materialized view `MV_PROD_TIME` which holds a hierarchical cube of sales aggregated across `TIMES` and `PRODUCTS` . Along with the materialized view, you will create a view on top of it. Also, you will create a view of the time dimension to use as an edge of the cube. The time edge will be partition outer joined to the sparse data in the materialized view.

For more information regarding hierarchical cubes, see the *Data Warehousing Reference Guide* , Chapter 19, "SQL for Aggregation in Data Warehouses".

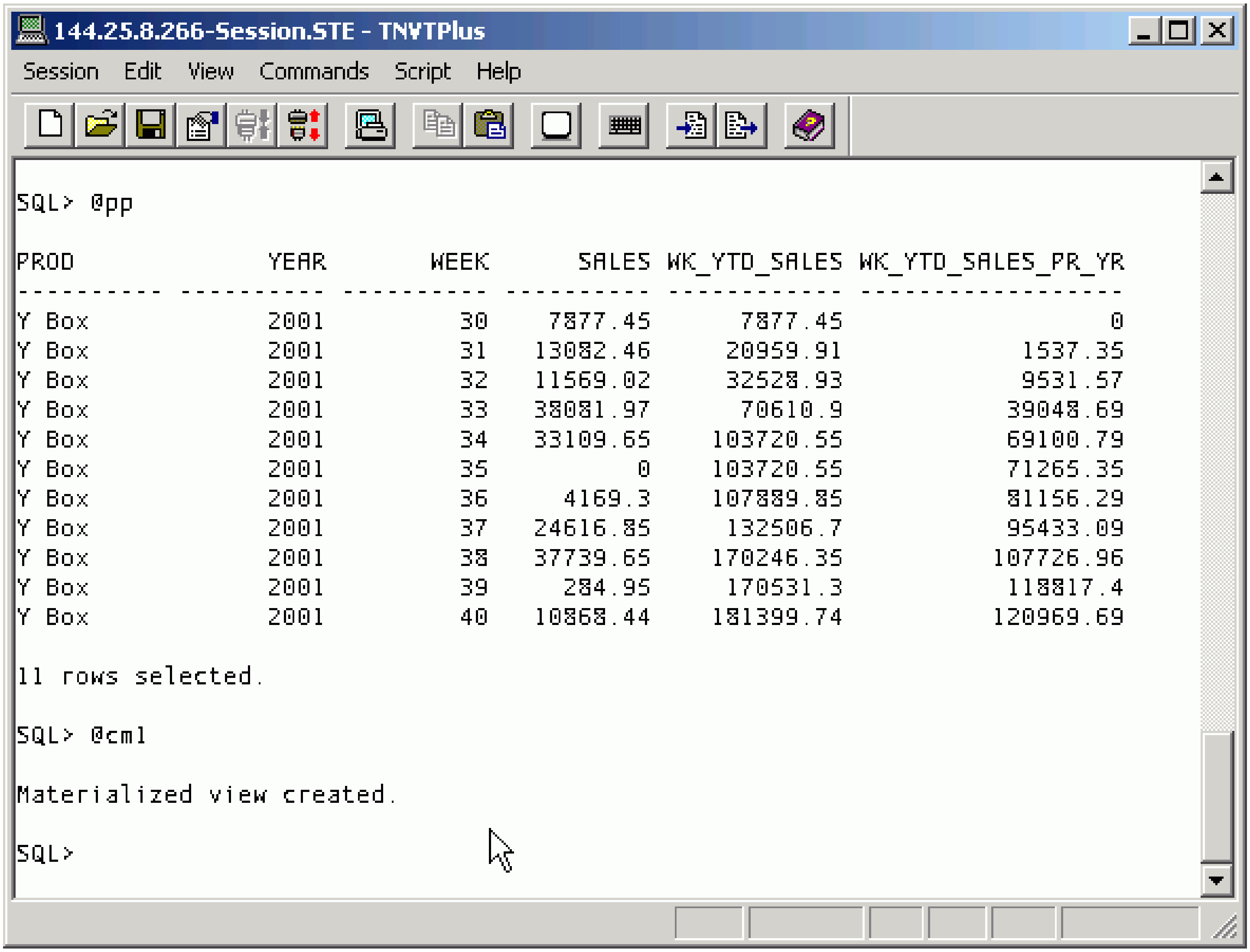
☐ **1.** You will create the materialized view. Note that the query is limited to just two products to keep processing time short. Execute the following SQL*Plus script:

```
@cm1
```

The **`cm1.sql`** script contains the following:

```
CREATE MATERIALIZED VIEW mv_prod_time
REFRESH COMPLETE ON DEMAND
AS
SELECT
  (CASE
   WHEN ((GROUPING(calendar_year)=0 )
     AND (GROUPING(calendar_quarter_desc)=1 ))
```

```
         THEN (TO_CHAR(calendar_year) || '_0')
         WHEN ((GROUPING(calendar_quarter_desc)=0 )
           AND (GROUPING(calendar_month_desc)=1 ))
         THEN (TO_CHAR(calendar_quarter_desc) || '_1')
         WHEN ((GROUPING(calendar_month_desc)=0 )
           AND (GROUPING(t.time_id)=1 ))
         THEN (TO_CHAR(calendar_month_desc) || '_2')
         ELSE (TO_CHAR(t.time_id) || '_3') END) Hierarchical_Time,
        calendar_year year,
        calendar_quarter_desc quarter,
        calendar_month_desc month,
        t.time_id day,
        prod_category cat,
        prod_subcategory subcat,
        p.prod_id prod,
        GROUPING_ID(prod_category, prod_subcategory, p.prod_id,
          calendar_year, calendar_quarter_desc,
          calendar_month_desc,t.time_id) gid,
        GROUPING_ID(prod_category, prod_subcategory, p.prod_id)  gid_p,
        GROUPING_ID(calendar_year, calendar_quarter_desc,
          calendar_month_desc, t.time_id) gid_t,
        SUM(amount_sold) s_sold,
        COUNT(amount_sold) c_sold,
        COUNT(*) cnt
      FROM SALES s, TIMES t, PRODUCTS p
      WHERE s.time_id = t.time_id AND
        p.prod_name in ('Bounce', 'Y Box')   AND
        s.prod_id = p.prod_id
      GROUP BY
        ROLLUP(calendar_year, calendar_quarter_desc,
               calendar_month_desc, t.time_id),
        ROLLUP(prod_category, prod_subcategory, p.prod_id);
```



```
SQL> @pp

PROD             YEAR       WEEK      SALES WK_YTD_SALES WK_YTD_SALES_PR_YR
---------- ---------- ---------- ---------- ------------ ------------------
Y Box            2001         30    7877.45      7877.45                  0
Y Box            2001         31   13082.46     20959.91            1537.35
Y Box            2001         32   11569.02     32528.93            9531.57
Y Box            2001         33   38081.97      70610.9           39048.69
Y Box            2001         34   33109.65    103720.55           69100.79
Y Box            2001         35          0    103720.55           71265.35
Y Box            2001         36     4169.3    107889.85           81156.29
Y Box            2001         37   24616.85     132506.7           95433.09
Y Box            2001         38   37739.65    170246.35          107726.96
Y Box            2001         39     284.95     170531.3           118817.4
Y Box            2001         40   10868.44    181399.74          120969.69

11 rows selected.

SQL> @cm1

Materialized view created.

SQL>
```

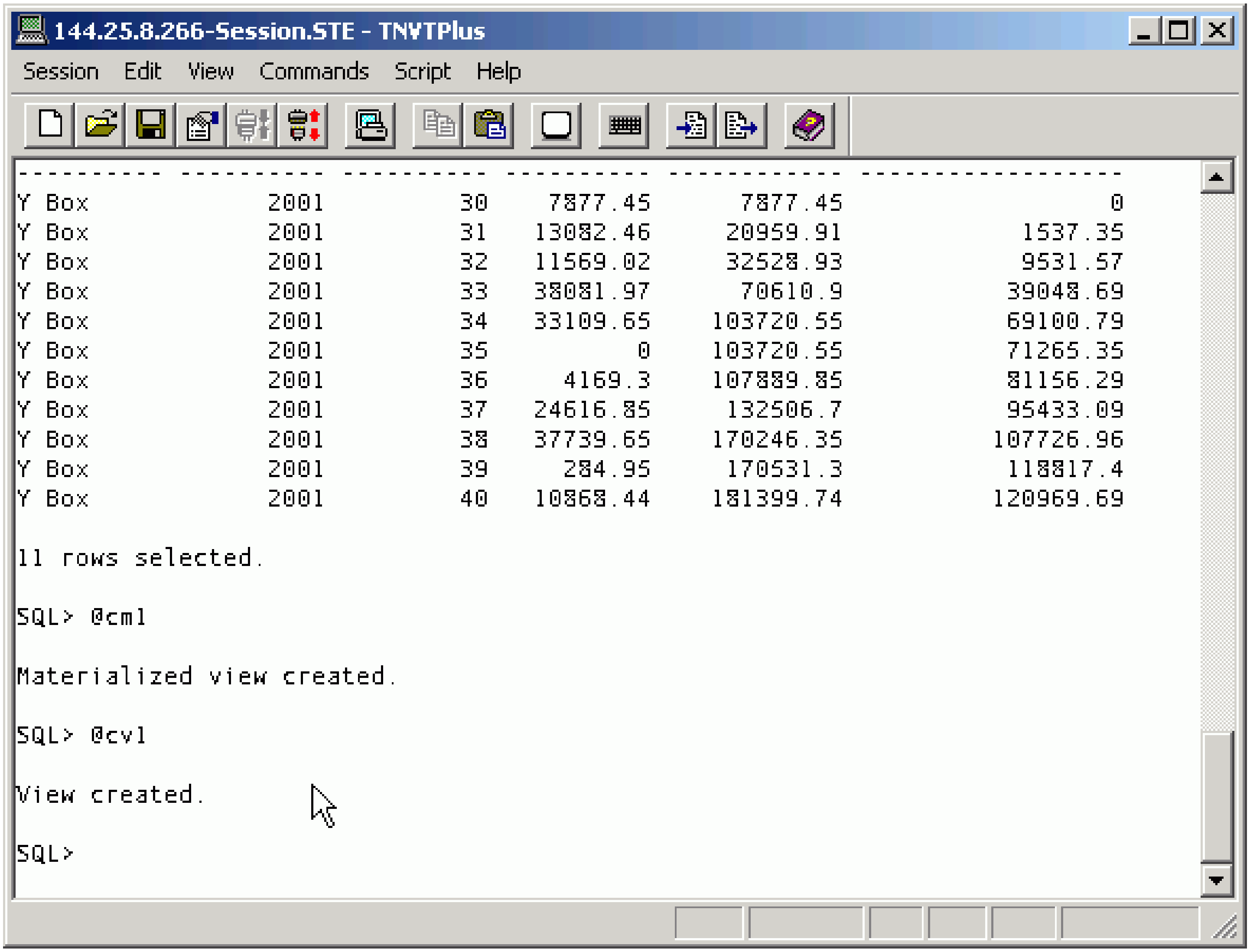
Since the materialized view is limited to two products, it has just over 2200 rows. Note that the column `Hierarchical_Time` contains string representations of time from all levels of the time hierarchy. The `CASE` expression used for the `Hierarchical_Time` column appends a marker (`_0` , `_1` , ...) to each date string to denote the time level of the value. A `_0` represents the year level, `_1` is quarters, `_2` is months, and `_3` is day. Note that the `GROUP BY` clause is a concatenated `ROLLUP` which specifies the rollup hierarchy for the time and product dimensions. The `GROUP BY` clause is what determines the hierarchical cube contents.

☐ **2.** Create a view `CUBE_PROD_TIME` with the same definition as the materialized view `MV_PROD_TIME` . Execute the following SQL*Plus script:

```
@cv1
```

The **`cv1.sql`** script contains the following:

```
CREATE OR REPLACE VIEW cube_prod_time
  AS
  SELECT
    (CASE
     WHEN ((GROUPING(calendar_year)=0 )
       AND (GROUPING(calendar_quarter_desc)=1 ))
     THEN (TO_CHAR(calendar_year) || '_0')
     WHEN ((GROUPING(calendar_quarter_desc)=0 )
       AND (GROUPING(calendar_month_desc)=1 ))
     THEN (TO_CHAR(calendar_quarter_desc) || '_1')
     WHEN ((GROUPING(calendar_month_desc)=0 )
       AND (GROUPING(t.time_id)=1 ))
     THEN (TO_CHAR(calendar_month_desc) || '_2')
     ELSE (TO_CHAR(t.time_id) || '_3') END) Hierarchical_Time,
     calendar_year year,
     calendar_quarter_desc quarter,
     calendar_month_desc month,
     t.time_id day,
     prod_category cat,
     prod_subcategory subcat,
     p.prod_id prod,
  GROUPING_ID(prod_category, prod_subcategory, p.prod_id,
    calendar_year, calendar_quarter_desc, calendar_month_desc,
    t.time_id) gid,
  GROUPING_ID(prod_category, prod_subcategory, p.prod_id) gid_p,
  GROUPING_ID(calendar_year, calendar_quarter_desc,
    calendar_month_desc, t.time_id) gid_t,
    SUM(amount_sold) s_sold,
    COUNT(amount_sold) c_sold,
    COUNT(*) cnt

  FROM SALES s, TIMES t, PRODUCTS p
  WHERE s.time_id = t.time_id AND
    p.prod_name IN ('Bounce', 'Y Box') AND
    s.prod_id = p.prod_id
  GROUP BY
  ROLLUP(calendar_year, calendar_quarter_desc,
         calendar_month_desc, t.time_id),
  ROLLUP(prod_category, prod_subcategory, p.prod_id);
```



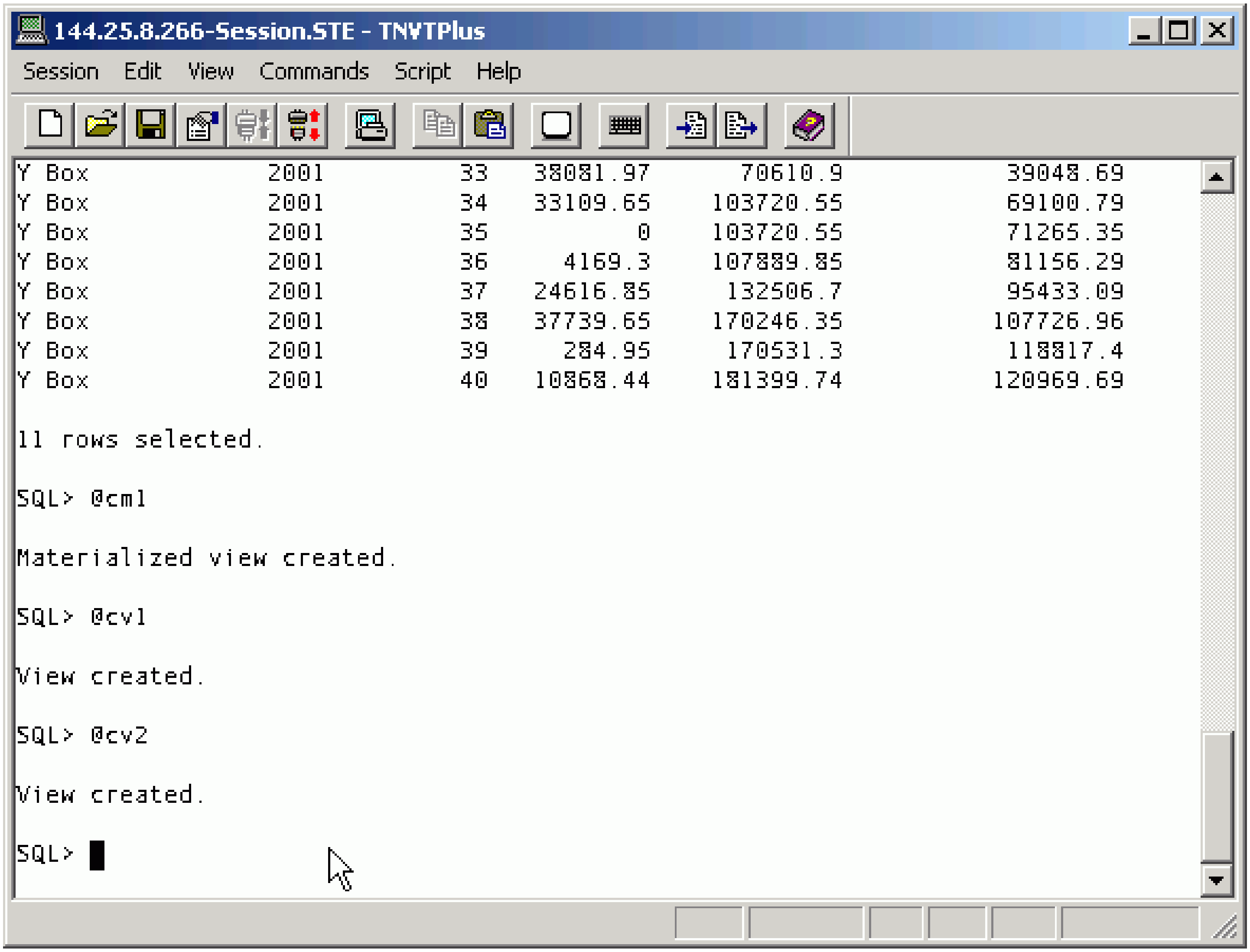
```
---------- ---------- ---------- ---------- ------------ ------------------
Y Box            2001         30    7877.45      7877.45                  0
Y Box            2001         31   13082.46     20959.91            1537.35
Y Box            2001         32   11569.02     32528.93            9531.57
Y Box            2001         33   38081.97      70610.9           39048.69
Y Box            2001         34   33109.65    103720.55           69100.79
Y Box            2001         35          0    103720.55           71265.35
Y Box            2001         36     4169.3    107889.85           81156.29
Y Box            2001         37   24616.85     132506.7           95433.09
Y Box            2001         38   37739.65    170246.35          107726.96
Y Box            2001         39     284.95     170531.3           118817.4
Y Box            2001         40   10868.44    181399.74          120969.69

11 rows selected.

SQL> @cm1

Materialized view created.

SQL> @cv1

View created.

SQL>
```

☐ **3.** You will create a view `EDGE_TIME` which is a complete set of date values. `EDGE_TIME` is the source for filling time gaps with a partitioned outer join. The column `HIERARCHICAL_TIME` in `EDGE_TIME` will be used in a partitioned join with the `HIERARCHICAL_TIME` column in the view `CUBE_PROD_TIME` .Execute the following SQL*Plus script:

```
@cv2
```

The **`cv2.sql`** script contains the following:

```
CREATE OR REPLACE VIEW edge_time
AS
SELECT
  (CASE
   WHEN ((GROUPING(calendar_year)=0 )
     AND (GROUPING(calendar_quarter_desc)=1 ))
   THEN (TO_CHAR(calendar_year) || '_0')
   WHEN ((GROUPING(calendar_quarter_desc)=0 )
     AND (GROUPING(calendar_month_desc)=1 ))
   THEN (TO_CHAR(calendar_quarter_desc) || '_1')
   WHEN ((GROUPING(calendar_month_desc)=0 )
     AND (GROUPING(time_id)=1 ))
   THEN (TO_CHAR(calendar_month_desc) || '_2')
   ELSE (TO_CHAR(time_id) || '_3') END) Hierarchical_Time,
   calendar_year yr,
   calendar_quarter_number qtr_num,
   calendar_quarter_desc qtr,
   calendar_month_number mon_num,
   calendar_month_desc mon,
   time_id - TRUNC(time_id, 'YEAR') + 1 day_num,
   time_id day,
   GROUPING_ID(calendar_year, calendar_quarter_desc,
     calendar_month_desc, time_id) gid_t
FROM TIMES
GROUP BY ROLLUP
(calendar_year,
(calendar_quarter_desc, calendar_quarter_number),
(calendar_month_desc, calendar_month_number), time_id);
```



```
Y Box                2001          33   38081.97       70610.9              39048.69
Y Box                2001          34   33109.65     103720.55              69100.79
Y Box                2001          35          0     103720.55              71265.35
Y Box                2001          36     4169.3     107889.85              81156.29
Y Box                2001          37   24616.85      132506.7              95433.09
Y Box                2001          38   37739.65     170246.35             107726.96
Y Box                2001          39     284.95      170531.3              118817.4
Y Box                2001          40   10868.44     181399.74             120969.69

11 rows selected.

SQL> @cm1

Materialized view created.

SQL> @cv1

View created.

SQL> @cv2

View created.

SQL> 
```

☐ **4.** You now have the required elements for the comparison query. You can obtain period-to-period comparison calculations at all time levels. It requires applying analytic functions to ahierarchical cube with dense data along the time dimension. Some of the calculations we can achieve for each time level are:

- sum of sales for prior period at all levels of time
- variance in sales over prior period
- sum of sales in the same period a year ago at all levels of time
- variance in sales over the same period last year

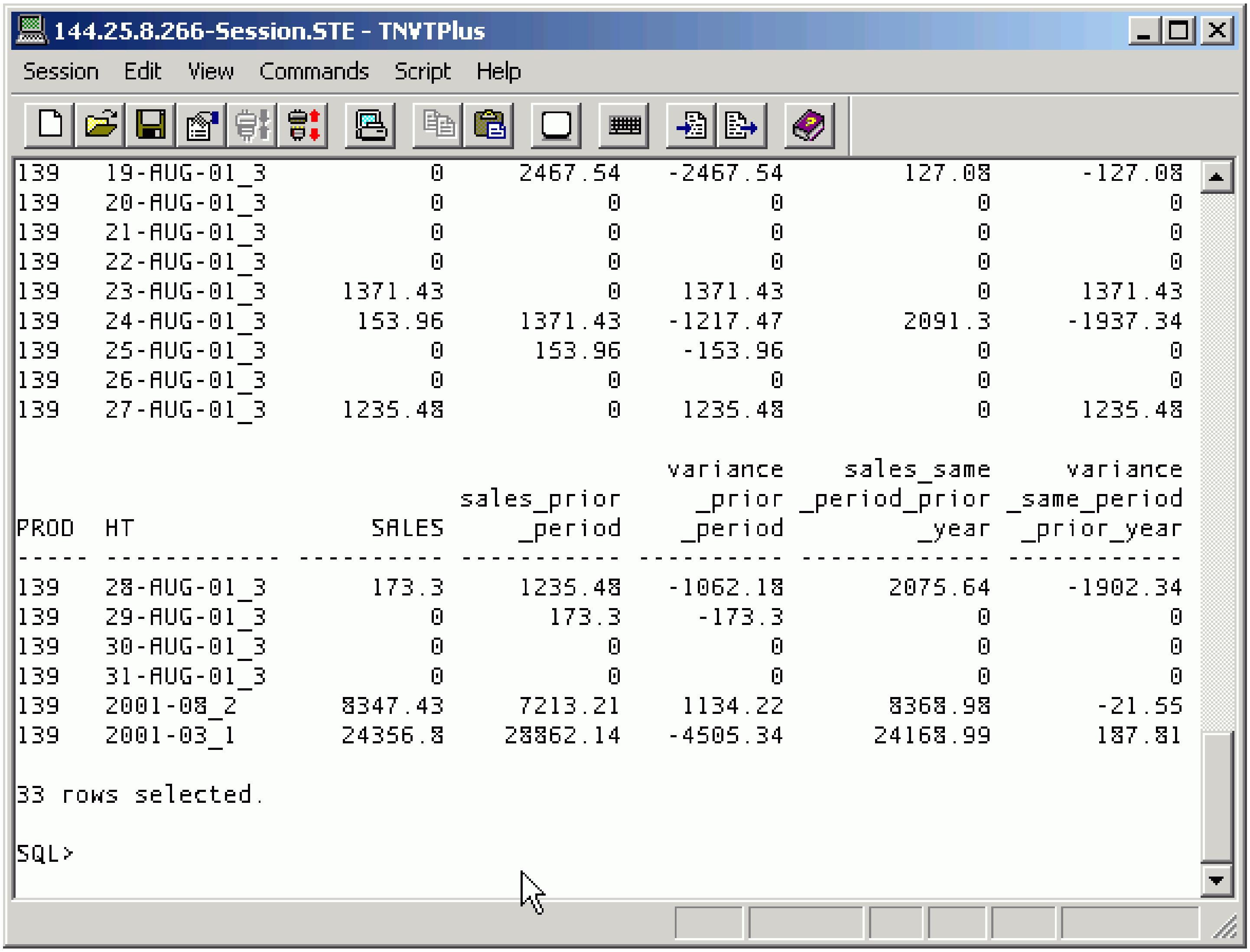
The following example performs all four of these calculations. It uses a partitioned outer join of the views `CUBE_PROD_TIME` and `EDGE_TIME` to create an in-line view of dense data called `DENSE_CUBE_PROD_TIME`. The query then uses the `LAG` function in the same way as the prior single-level example. The outer `WHERE` clause specifies time at three levels: the days of August 2001, the entire month, and the entire third quarter of 2001. Note that the last two rows of the results contain the month level and quarter level aggregations.

Execute the following SQL*Plus script:

```
@mt
```

The **`mt.sql`** script contains the following:

```
SELECT
  substr(prod,1,4) prod, substr(Hierarchical_Time,1,12) ht,
  sales,
  sales_prior_period,
  sales - sales_prior_period variance_prior_period,
  sales_same_period_prior_year,
  sales - sales_same_period_prior_year variance_same_period_p_year
FROM
  (SELECT cat, subcat, prod, gid_p, gid_t, Hierarchical_Time,
     yr, qtr, mon, day, sales,
     LAG(sales, 1) OVER (PARTITION BY gid_p, cat, subcat, prod,
     gid_t ORDER BY yr, qtr, mon, day)
     sales_prior_period,
     LAG(sales, 1) OVER (PARTITION BY gid_p, cat, subcat, prod,
     gid_t, qtr_num, mon_num, day_num ORDER BY yr)
     sales_same_period_prior_year
   FROM
    (SELECT c.gid, c.cat, c.subcat, c.prod, c.gid_p,
     t.gid_t, t.yr, t.qtr, t.qtr_num, t.mon, t.mon_num,
     t.day, t.day_num, t.Hierarchical_Time, NVL(s_sold,0) sales
     FROM cube_prod_time c
     PARTITION BY (gid_p, cat, subcat, prod)
     RIGHT OUTER JOIN edge_time t
    ON ( c.gid_t = t.gid_t AND c.Hierarchical_Time = t.Hierarchical_Time)
    ) dense_cube_prod_time
    )          -- side by side current,prior and prior year sales
WHERE prod IN (139) AND gid_p=0 AND -- 1 product and product level data
  ( (mon IN ('2001-08' ) AND gid_t IN (0, 1) ) OR -- day and month data
  ( qtr IN ('2001-03' ) AND gid_t IN (3) ) ) -- quarter level data
ORDER BY day;
```



```
139   19-AUG-01_3           0     2467.54   -2467.54        127.08      -127.08
139   20-AUG-01_3           0           0          0             0            0
139   21-AUG-01_3           0           0          0             0            0
139   22-AUG-01_3           0           0          0             0            0
139   23-AUG-01_3     1371.43           0    1371.43             0      1371.43
139   24-AUG-01_3      153.96     1371.43   -1217.47        2091.3     -1937.34
139   25-AUG-01_3           0      153.96    -153.96             0            0
139   26-AUG-01_3           0           0          0             0            0
139   27-AUG-01_3     1235.48           0    1235.48             0      1235.48

                                            variance    sales_same     variance
                              sales_prior     _prior _period_prior _same_period
PROD  HT                SALES     _period    _period         _year  _prior_year
----- ------------ ---------- ----------- ---------- ------------- ------------
139   28-AUG-01_3       173.3     1235.48   -1062.18       2075.64     -1902.34
139   29-AUG-01_3           0       173.3     -173.3             0            0
139   30-AUG-01_3           0           0          0             0            0
139   31-AUG-01_3           0           0          0             0            0
139   2001-08_2       8347.43     7213.21    1134.22       8368.98       -21.55
139   2001-03_1       24356.8    28862.14   -4505.34      24168.99       187.81

33 rows selected.

SQL>
```

Example of Creating a Custom Member in a Dimension

Back to List of Topics

In many OLAP tasks, it is helpful to define custom members in a dimension. For instance, you might define a specialized time period for analyses. You can use a partitioned outer join to temporarily add a member to a dimension. Note that the new SQL `MODEL` clause introduced in Oracle Database 10 *g* is suitable for creating more complex scenarios involving new members in dimensions. See the *Data Warehousing Reference Guide* Chapter 22, "SQL for Modeling" for more information on this topic.

In this exercise, you will define a new member for the `TIME` dimension. You will create a 13th member of the Month level in the `TIME` dimension. This 13th month is defined as the summation of the sales for each product in the first month of each quarter of year 2001. You will build this solution using the views and tables created in the prior example.

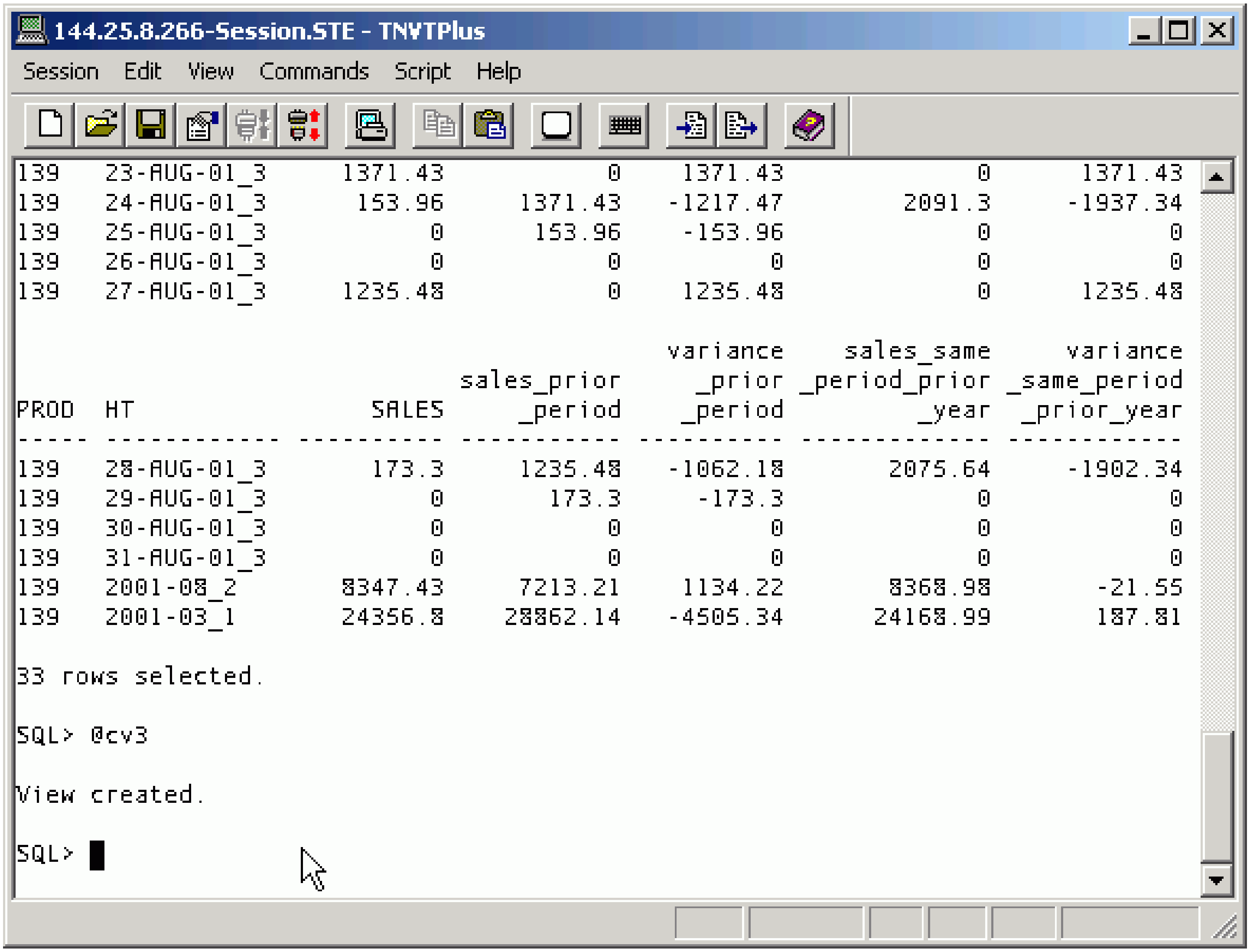
☐ **1.** Create a view with the new member added to the appropriate dimension. The view uses a `UNION ALL` operation to add the new member. To query using the custom member, use a `CASE` expression and a partitioned outer join. Execute the following SQL*Plus script:

```
@cv3
```

The **`cv3.sql`** script contains the following:

```
CREATE OR REPLACE VIEW time_c  AS
   (SELECT *
      FROM edge_time
   UNION ALL
   SELECT '2001-13_2', 2001, 5, '2001-05', 13, '2001-13', null, null,
     8 -- <gid_of_mon>
     FROM DUAL);
```

In the statement shown, the view `TIME_C` is defined by performing a `UNION ALL` of the `EDGE_TIME` view (defined in the prior example) and the user-defined 13th month. The `UNION ALL` specifies the attributes for a 13th month member by doing a `SELECT` from the `DUAL` table. Note that the grouping `id` , column `gid_t` , is set to 8, and the quarter number is set to `5` .



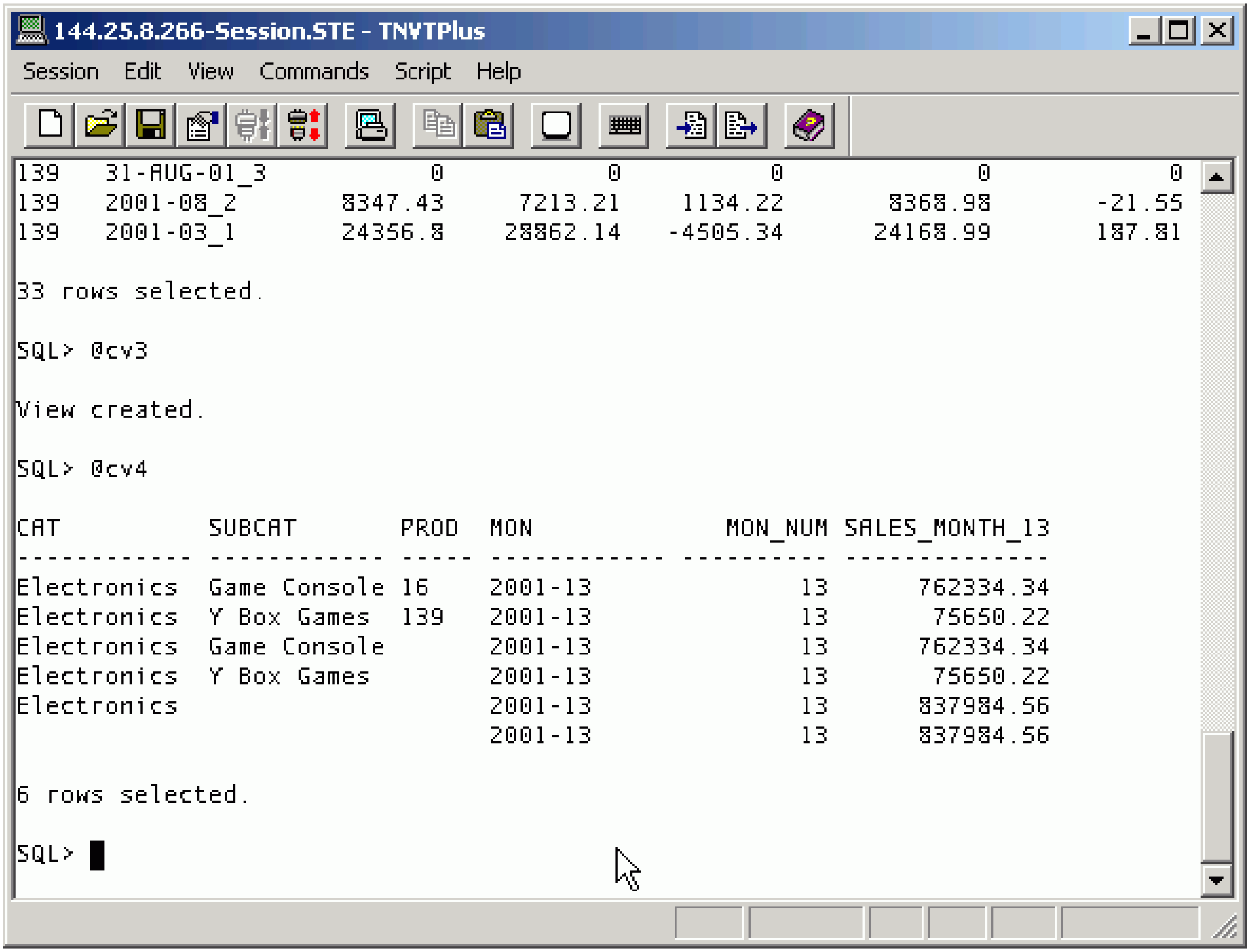
```
139   23-AUG-01_3     1371.43            0    1371.43           0       1371.43
139   24-AUG-01_3      153.96      1371.43   -1217.47      2091.3      -1937.34
139   25-AUG-01_3           0       153.96    -153.96           0             0
139   26-AUG-01_3           0            0          0           0             0
139   27-AUG-01_3     1235.48            0    1235.48           0       1235.48

                                              variance    sales_same      variance
                                sales_prior      _prior _period_prior _same_period
PROD  HT                  SALES     _period     _period         _year  _prior_year
----- ------------ ---------- ----------- ---------- ------------- ------------
139   28-AUG-01_3       173.3     1235.48   -1062.18       2075.64     -1902.34
139   29-AUG-01_3           0       173.3     -173.3             0            0
139   30-AUG-01_3           0           0          0             0            0
139   31-AUG-01_3           0           0          0             0            0
139   2001-08_2       8347.43     7213.21    1134.22       8368.98       -21.55
139   2001-03_1       24356.8    28862.14   -4505.34      24168.99       187.81

33 rows selected.

SQL> @cv3

View created.

SQL> 
```

☐ **2.** The in-line view of the query shown below performs a partitioned outer join of `CUBE_PROD_TIME` with `TIME_C` . This step creates sales data for the 13th month at each level of product aggregation. In the main query, the analytic function `SUM` is used with a `CASE` expression to compute the 13th month, which is defined as the summation of the first month's sales of each quarter. Execute the following SQL*Plus script:

```
@cv4
```

The **`cv4.sql`** script contains the following:

```
SELECT * from
(
SELECT substr(cat,1,12) cat, substr(subcat,1,12)  subcat,
       substr(prod,1,9) prod,  mon, mon_num,
       SUM(CASE WHEN mon_num IN (1, 4, 7, 10)
                THEN s_sold
                ELSE NULL
                END)
           OVER (PARTITION BY gid_p, prod, subcat, cat, yr) sales_month_13
FROM
   (SELECT c.gid, c.prod, c.subcat, c.cat,  gid_p,
        t.gid_t , t.day, t.mon, t.mon_num,
        t.qtr, t.yr, NVL(s_sold,0) s_sold
    FROM cube_prod_time c
    PARTITION BY   (gid_p, prod, subcat, cat)
    RIGHT OUTER JOIN time_c  t ON
      (c.gid_t = t.gid_t AND c.Hierarchical_Time = t.Hierarchical_Time)
   )
)
WHERE  mon_num=13;
```



```
139   31-AUG-01_3           0           0          0             0            0
139   2001-08_2       8347.43     7213.21    1134.22       8368.98       -21.55
139   2001-03_1       24356.8    28862.14   -4505.34      24168.99       187.81

33 rows selected.

SQL> @cv3

View created.

SQL> @cv4

CAT          SUBCAT       PROD  MON             MON_NUM SALES_MONTH_13
------------ ------------ ----- ------------ ---------- --------------
Electronics  Game Console 16    2001-13              13      762334.34
Electronics  Y Box Games  139   2001-13              13       75650.22
Electronics  Game Console       2001-13              13      762334.34
Electronics  Y Box Games        2001-13              13       75650.22
Electronics                     2001-13              13      837984.56
                                2001-13              13      837984.56

6 rows selected.

SQL> █
```

The SUM function used in generating these results had a CASE statement to limit the data to months 1 , 4 , 7 and 10 within each year. Due to the tiny data set, with just 2 products, the rollup values of the results are necessarily repetitions of lower level aggregations. For a more realistic set of rollup values, you can include more products from the "Game Console" and "Y Box Games" subcategories in the underlying materialized view.