Using the SQL `MODEL` Clause to Define Inter-row Calculations

Purpose

In this module you learn how to use the Oracle Database 10 *g* SQL `MODEL` clause to perform inter-row calculations.

Topics

This module will discuss the following:

- Overview
- Prerequisites
- Sample Data
- Example Syntax
- Positional and Symbolic Cell References
- The `CV( )` Function and ANY Wildcard
- `FOR` Loops
- Order of Evaluation of Rules
- Reference `MODEL` s
- Iterative `MODEL` s
- Ordered Rules

Overview

Back to List of Topics

Oracle Database 10 *g* SQL `MODEL` Clause Overview

With the SQL `MODEL` clause, you can define a multidimensional array on query results and then apply rules on the array to calculate new values. The rules can be sophisticated interdependent calculations. By integrating advanced calculations into the database, performance, scalability and manageability are enhanced significantly compared to external solutions. Rather than copying data into separate applications or PC spreadsheets, users can keep their data within the Oracle environment.

The `MODEL` clause defines a multidimensional array by mapping the columns of a query into three groups: partitioning, dimension, and measure columns. These elements perform the following tasks:

- Partitions define logical blocks of the result set in a way similar to the partitions of the analytical functions (described in the Data Warehousing Guide Chapter 21, "SQL for Analysis in Data Warehouses"). `MODEL` rules are applied to the cells of each partition.

- Dimensions identify each measure cell within a partition. These columns are identifying characteristics such as date, region and product name.

- Measures are analogous to the measures of a fact table in a star schema. They typically contain numeric values such as sales units or cost. Each cell is accessed within its partition by specifying its full combination of dimensions.

To create rules on these multidimensional arrays, you define computation rules expressed in terms of the dimension values. The rules are flexible and concise, and can use wild cards and `FOR` loops for maximum expressiveness. Calculations built with the `MODEL` clause improve on traditional spreadsheet calculations by integrating analyses into the database, improving readability with symbolic referencing, and providing scalability and much better manageability.

The figure below gives a conceptual overview of the model feature using a hypothetical sales table. The table has columns for country, product, year and sales amount. The figure has three parts. The top segment shows the concept of dividing the table into partitioning, dimension and measure columns. The middle segment shows two hypothetical rules that forecast sales for `Prod1` and `Prod2` as the calculated value of product sales from the two previous years. Finally, the third part shows the output of a query applying the rules to such a table with hypothetical data. The black output is data retrieved from the database, while the blue output shows rows calculated from rules. Note that the rules are applied within each partition.

Columns mapped to Partition, Dimension and Measure

COUNTRY	PRODUCT	YEAR	SALES
Partition	Dimension	Dimension	Measure

Rules:

```
sales('prod1', 2002) = sales('prod1', 2000) + sales('prod1', 2001)
sales('prod2', 2002) = sales('prod2', 2000) + sales('prod2', 2001)
```

Output of the `MODEL` clause:

COUNTRY	PRODUCT	YEAR	SALES
Partition	Dimension	Dimension	Measure
A	prod1	2000	10
A	prod1	2001	15
A	prod2	2000	12

A	prod2	2001	16
B	prod1	2000	21
B	prod1	2001	23
B	prod2	2000	28
B	prod2	2001	29
A	**prod1**	**2002**	**25**
A	**prod2**	**2002**	**28**
B	**prod1**	**2002**	**44**
B	**prod2**	**2002**	**57**

Note that the `MODEL` clause does not update existing data in tables, nor does it insert new data into tables: to change values in a table, the Model results must be supplied to an INSERT or UPDATE or MERGE statement.

Prerequisites

Back to List

Before starting this module, you should have:

☐ **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 and unzip model_clause.zip into your working directory (i.e. /home/oracle/wkdir)

Sample Data

Back to List of Topics

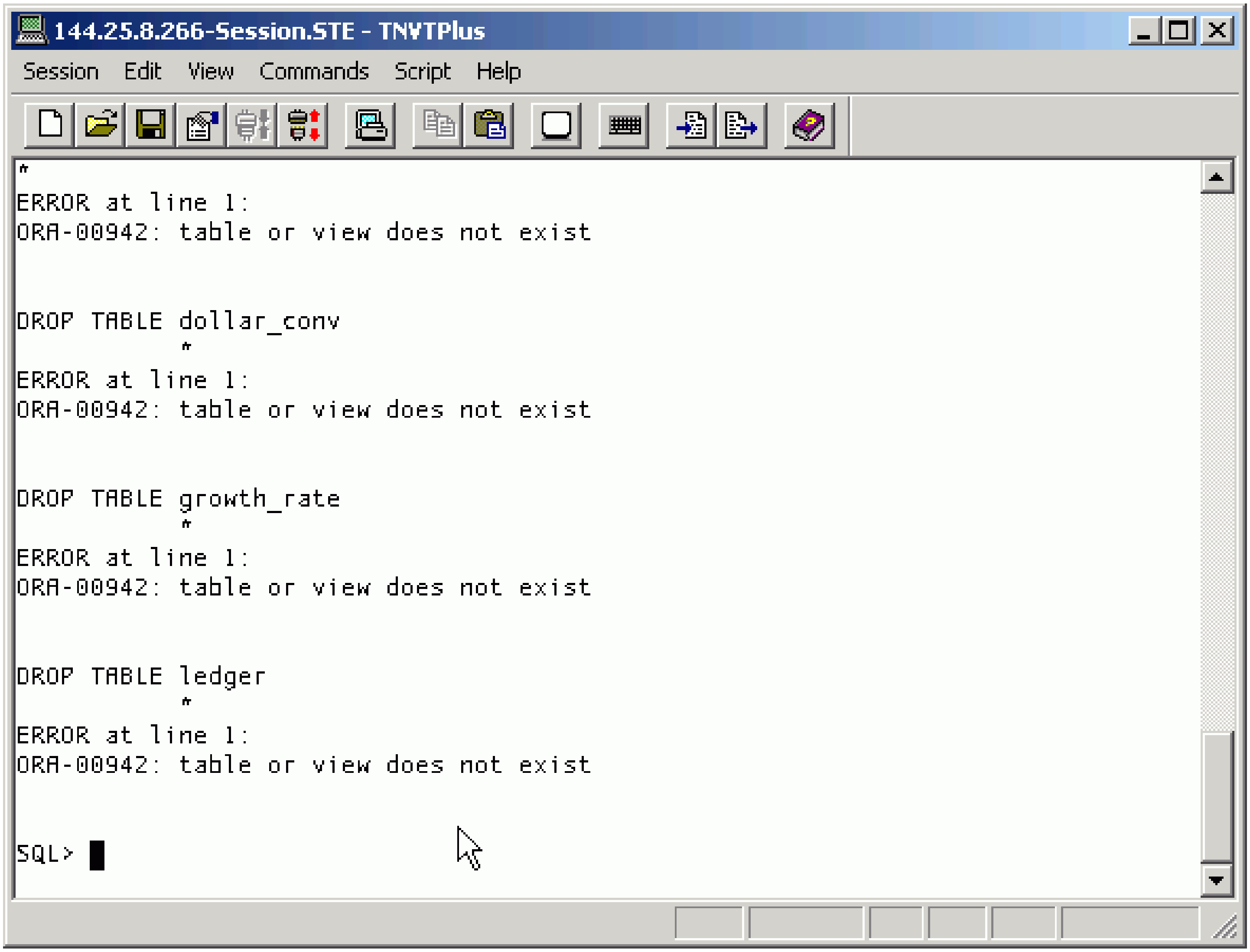
You will use the `SH` schema to create a view. This view provides annual sums for product sales, in dollars and units, by country, aggregated across all channels

☐ **1.** You will first make sure you have a clean environment. From a terminal window, execute the following command (s):

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

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

```
DROP VIEW sales_view;
DROP TABLE dollar_conv;
DROP TABLE growth_rate;
DROP TABLE ledger;
```


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

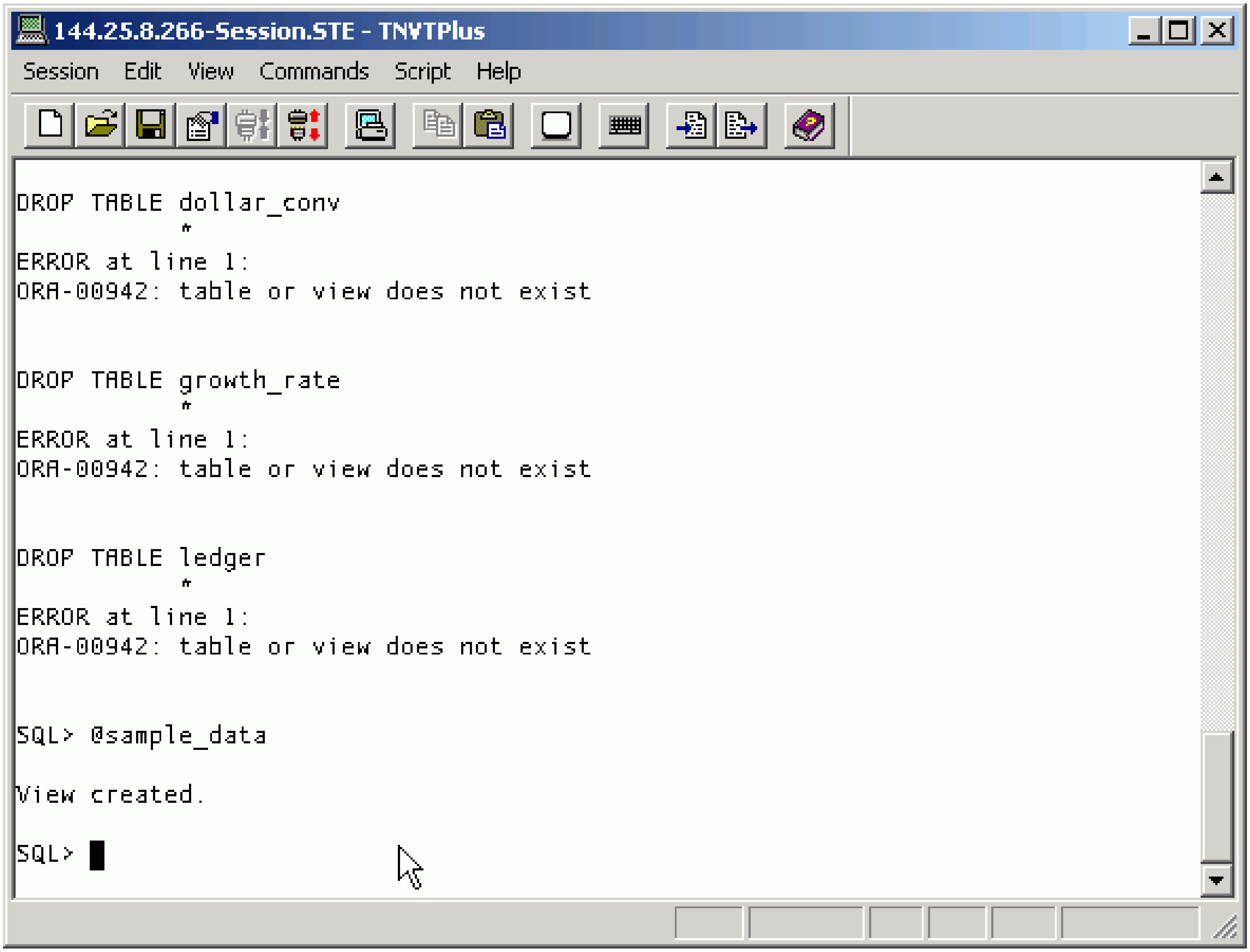
```
*
ERROR at line 1:
ORA-00942: table or view does not exist


DROP TABLE dollar_conv
           *
ERROR at line 1:
ORA-00942: table or view does not exist


DROP TABLE growth_rate
           *
ERROR at line 1:
ORA-00942: table or view does not exist


DROP TABLE ledger
           *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL>
```

☐ **2.** Now you can create the SALES_VIEW view. From your SQL*Plus session, execute the following script:

```
@sample_data
```

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

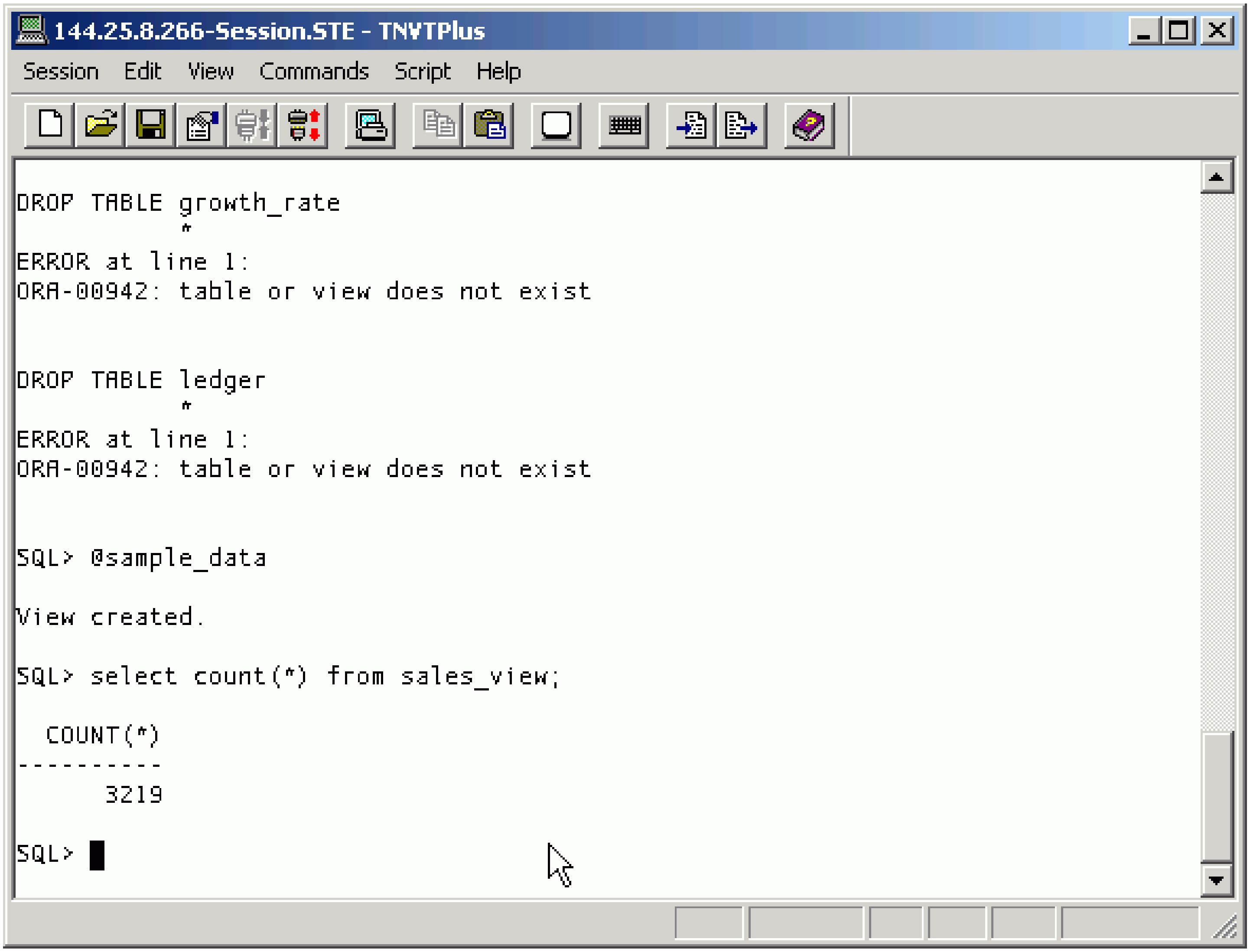
```
CREATE VIEW sales_view AS

   SELECT country_name country, prod_name prod, calendar_year year,

          SUM(amount_sold) sale, COUNT(amount_sold) cnt

   FROM   sales, times, customers, countries, products

   WHERE  sales.time_id = times.time_id AND

          sales.prod_id = products.prod_id

   AND    sales.cust_id = customers.cust_id

   AND    customers.country_id = countries.country_id

   GROUP BY country_name, prod_name, calendar_year

/
```


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

```
DROP TABLE dollar_conv
           *
ERROR at line 1:
ORA-00942: table or view does not exist


DROP TABLE growth_rate
           *
ERROR at line 1:
ORA-00942: table or view does not exist


DROP TABLE ledger
           *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL> @sample_data

View created.

SQL> 
```

☐ **3.** Verify the view is created correctly and that 3219 ro ws exist. From your SQL*Plus session, execute the following command:

```
SELECT COUNT(*) FROM sales_view;
```


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

```
DROP TABLE growth_rate
           *
ERROR at line 1:
ORA-00942: table or view does not exist


DROP TABLE ledger
           *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL> @sample_data

View created.

SQL> select count(*) from sales_view;

  COUNT(*)
----------
      3219

SQL>
```

☐ **4.** To maximize performance, your system should already have a materialized view built on the data that is used by the view above. The materialized view is created during the installation of the `sh` schema data. Oracle's summary management system will automatically rewrite any query using the view above so that it takes advantage of the materialized view.

Example Syntax

Back to List of Topics

As an initial example of models, consider the following statement:

```
SELECT SUBSTR(country,1,20) country,
       SUBSTR(prod,1,15) prod, year, sales
```

```
FROM sales_view
WHERE country IN ('Italy','Japan')

MODEL RETURN UPDATED ROWS
     PARTITION BY (country)
     DIMENSION BY (prod, year)
     MEASURES (sale sales)
     RULES (
       sales['Bounce', 2002] = sales['Bounce', 2001] + sales['Bounce', 2000],
       sales['Y Box', 2002] = sales['Y Box', 2001],
       sales['2_Products', 2002] = sales['Bounce', 2002] + sales['Y Box', 2002])
ORDER BY country, prod, year;
```

The results are:

```
COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                2_Products            2002   90387.54
Italy                Bounce                2002    9179.99
Italy                Y Box                 2002   81207.55
Japan                2_Products            2002  101071.96
Japan                Bounce                2002   11437.13
Japan                Y Box                 2002   89634.83
```

This statement partitions by country, so the rules are applied to data of one country at a time. Note that the data ends with 2001, so any rules defining values for 2002 or later will insert new cells. The first rule defines the sales of `Bounce` in 2002 as the sum of sales in 2000 and 2001. The second rule defines the sales for `Y Box` in 2002 as being the same value as they were for 2001. The third rule defines a category called `2_Products` , which is simply the sum of adding the 2002 `Bounce` and `Y Box` values together. Note that the values for `2_Products` are derived from the results of the two prior rules, so those rules must be executed before the `2_Products` rule.

Syntax Guidelines

- Note that the " `RETURN UPDATED ROWS` " clause following the keyword `MODEL` limits the results to just those rows that were created or updated in this query. Using this clause is a convenient way to limit result sets to just the newly calculated values. You will use the `RETURN UPDATED ROWS` clause throughout the examples.
- The keyword `RULES` , shown in the examples at the start of the rules, is optional, but recommended for easier reading.
- Many of our examples do not require `ORDER BY` on the `COUNTRY` column. It is included in the specification in case you want to modify the examples and add multiple countries.

Technical Details

The following examples move through the major features of the `MODEL` clause, building from basic cell references to reference models and iterative models.

Positional and Symbolic Cell References

Back to List of Topics

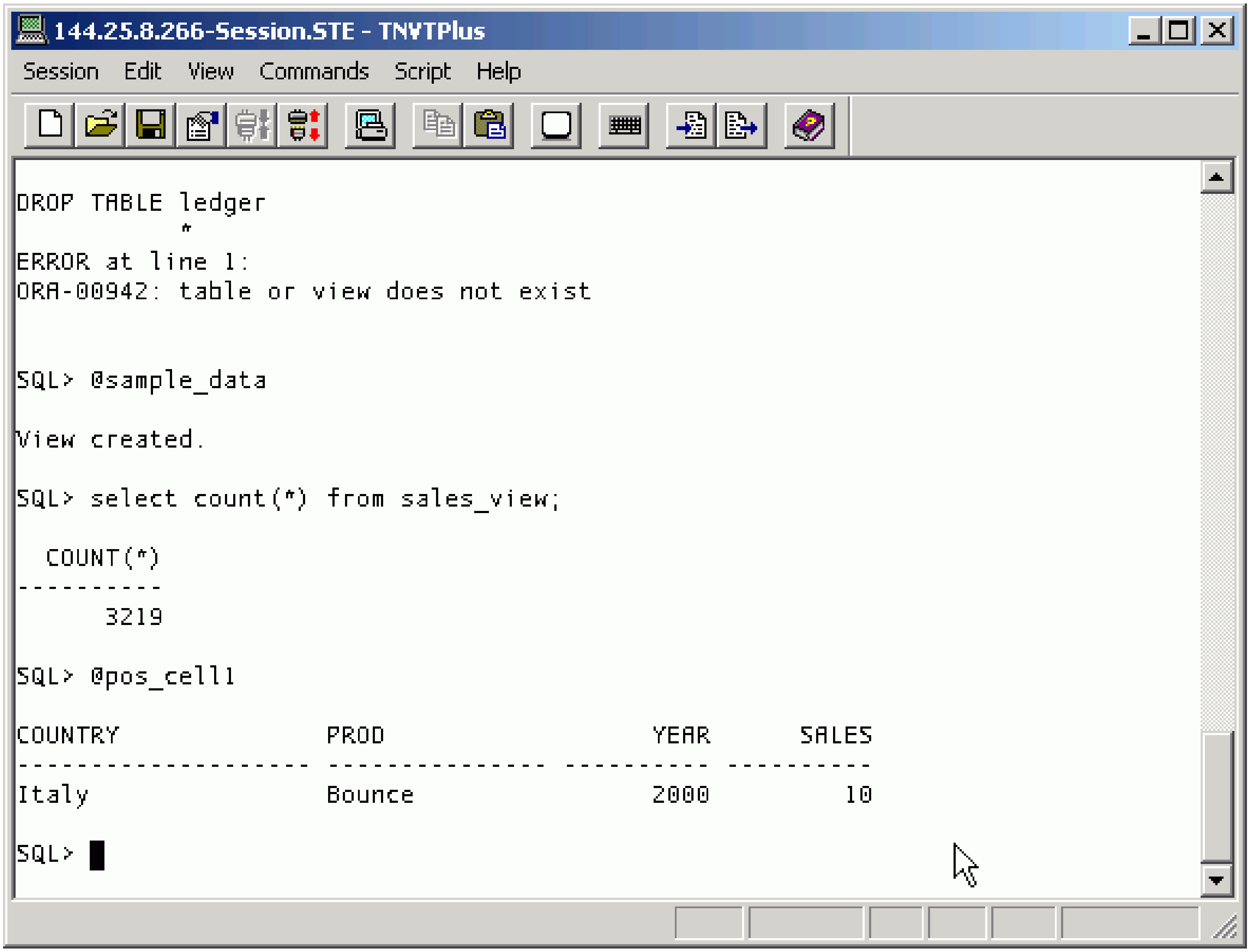
This section examines the techniques for symbolic and positional referencing cells in the `MODEL` statement.

☐ **1.** You want to view the `SALES` value for the product `Bounce` in the year 2000, in `Italy` , and set it to `10` . To do so, use a "positional cell reference". The value for the cell reference is matched to the appropriate dimension based on its position in the expression. The `DIMENSION BY` clause of the model determines the position assigned to each dimension: in this case, the first position is product (" `PROD` ") and the second position is `YEAR` . From your SQL*Plus session, execute the following script:

```
@pos_cell1
```

The **`pos_cell1.sql`** script contains the following:

```
SELECT SUBSTR(country,1,20) country,

       SUBSTR(prod,1,15) prod, year, sales

  FROM   sales_view

  WHERE   country='Italy'

  MODEL  RETURN UPDATED ROWS

    PARTITION BY (country)

    DIMENSION BY (prod, year)

    MEASURES (sale sales)

    RULES  (

      sales['Bounce', 2000] = 10 )

ORDER BY country, prod, year

/
```


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

```
DROP TABLE ledger
           *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL> @sample_data

View created.

SQL> select count(*) from sales_view;

  COUNT(*)
----------
      3219

SQL> @pos_cell1

COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce                2000         10

SQL> 
```

☐ **2.** You want to create a forecast value of SALES for the product `Bounce` in the year `2005` , in `Italy` , and set it to `20` . Use a rule in the SELECT statement that sets the year value to `2005` and thus create a new cell in the array. From your SQL*Plus session, execute the following script:

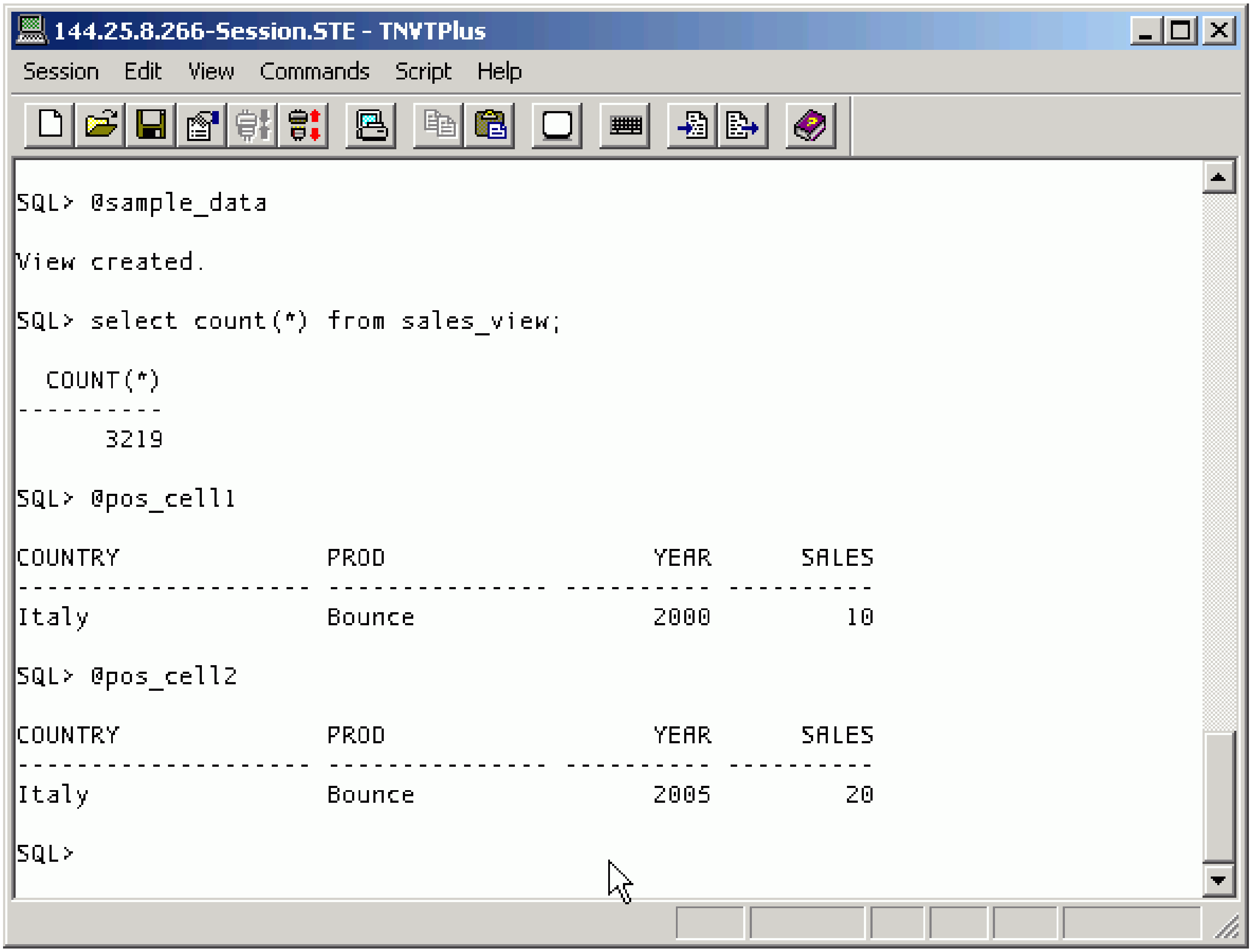
```
@pos_cell2
```

The **`pos_cell2.sql`** script contains the following:

```
SELECT SUBSTR(country,1,20) country,
       SUBSTR(prod,1,15) prod, year, sales
FROM   sales_view
```

```
WHERE  country='Italy'

  MODEL  RETURN UPDATED ROWS

    PARTITION BY (country)

    DIMENSION BY (prod, year)

    MEASURES (sale sales)

    RULES  (

      sales['Bounce', 2005] = 20 )

ORDER BY country, prod, year

/
```

Note: If you want to create new cells, such as values for future years, you must use positional references or `FOR` loops (discussed later in this lesson). That is, positional reference permits both updates and inserts into the array. This is called the `UPSERT` process.



```
SQL> @sample_data

View created.

SQL> select count(*) from sales_view;

  COUNT(*)
----------
      3219

SQL> @pos_cell1

COUNTRY                   PROD                       YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                     Bounce                     2000         10

SQL> @pos_cell2

COUNTRY                   PROD                       YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                     Bounce                     2005         20

SQL>
```

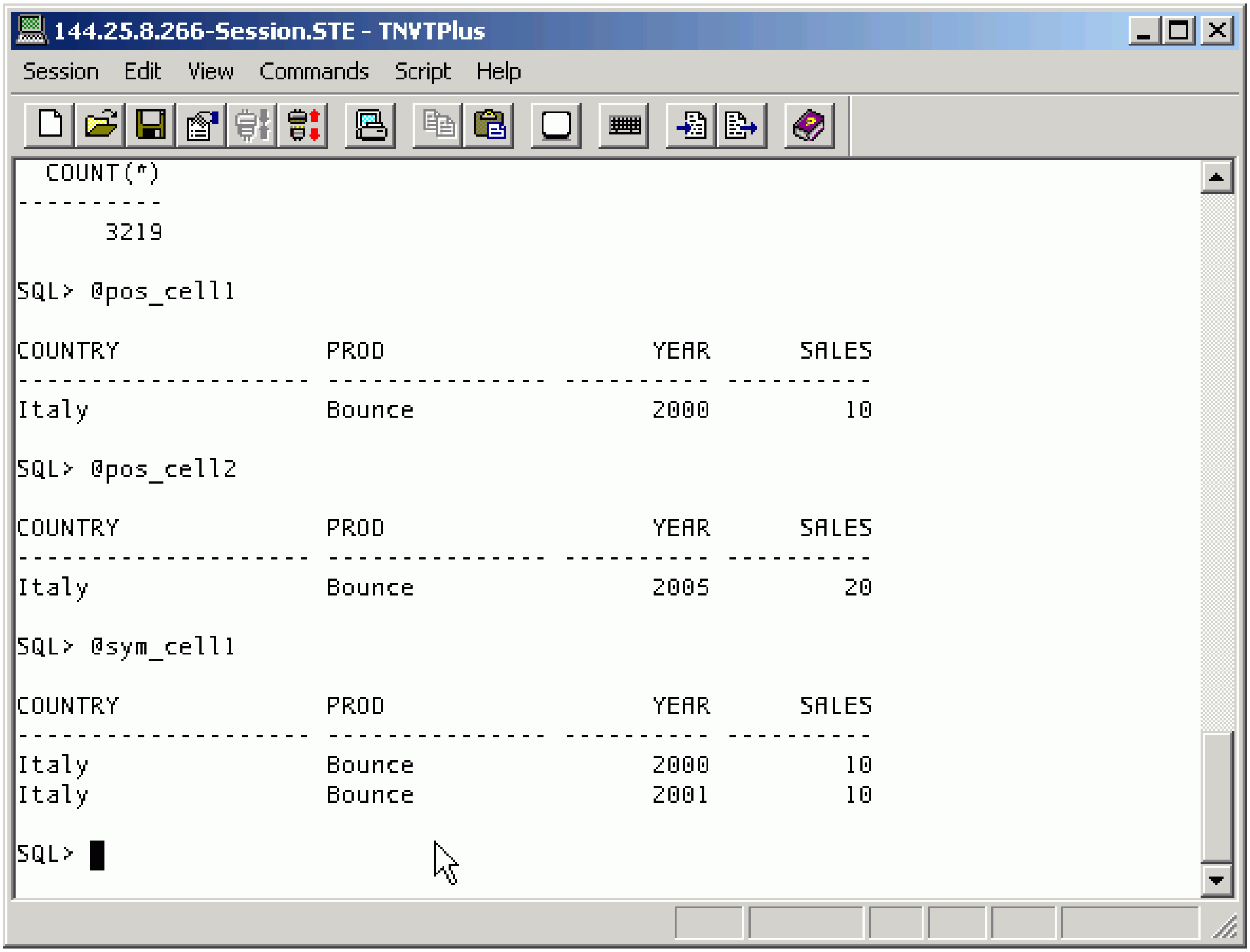
☐ **3.** You want to update the `SALES` for the product `Bounce` in all years after `1999` where the values are recorded for `Italy` and set them to `10` . To do so, use a "symbolic cell reference". The value for the cell reference is matched to the appropriate dimension using Boolean conditions. You can use all the normal operators such as `<` , `>` , `IN` , and `BETWEEN` . In this case the query looks for product value equal to `Bounce` and any year value greater than `1999` . This shows how a single rule can access multiple cells. From your SQL*Plus session, execute the following script:

```
@sym_cell1
```

The **`sym_cell1.sql`** script contains the following:

```
SELECT SUBSTR(country,1,20) country,

       SUBSTR(prod,1,15) prod, year, sales

FROM   sales_view

WHERE  country='Italy'

  MODEL   RETURN UPDATED ROWS

    PARTITION BY (country)

    DIMENSION BY (prod, year)

    MEASURES (sale sales)

    RULES (

      sales[prod='Bounce', year>1999] = 10 )

ORDER BY country, prod, year

/
```

Note: Symbolic references are very powerful, but they are solely for updating existing cells: they cannot create new cells such as sales projections in future years.



```
  COUNT(*)
----------
      3219

SQL> @pos_cell1

COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce                2000         10

SQL> @pos_cell2

COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce                2005         20

SQL> @sym_cell1

COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce                2000         10
Italy                Bounce                2001         10

SQL>
```

☐ **4.** You want a single query to update the sales for several products in several years for multiple countries, and you also want it to insert new cells. By placing several rules into one query, processing is more efficient since it reduces the number of times needed to access the data. It also allows for more concise SQL, supporting higher developer productivity. From your SQL*Plus session, execute the following script:

`@pos_sym`

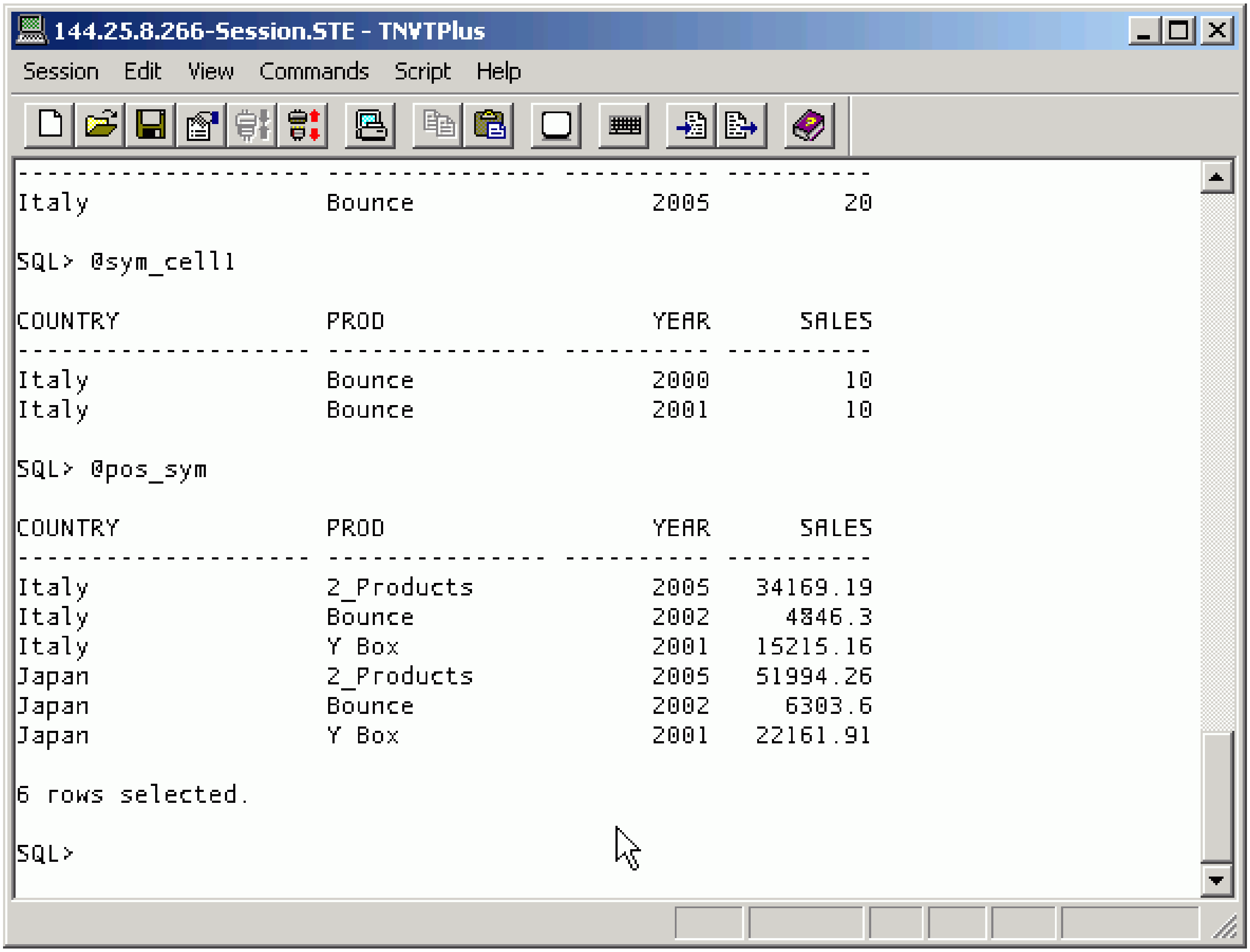
The **`pos_sym.sql`** script contains the following:

```
SELECT SUBSTR(country,1,20) country,

       SUBSTR(prod,1,15) prod, year,   sales

FROM   sales_view  WHERE country IN  ('Italy','Japan')
```

```
  MODEL  RETURN UPDATED ROWS

    PARTITION BY (country)

    DIMENSION BY (prod, year)

    MEASURES (sale sales)

    RULES (

      sales['Bounce', 2002] = sales['Bounce', year = 2001] ,

      --positional notation:  can insert new cell

      sales['Y Box', year>2000] = sales['Y Box', 1999],

      --symbolic notation: can update existing cell

      sales['2_Products', 2005] =

           sales['Bounce', 2001] + sales['Y Box', 2000]  )

      --positional notation:  permits insert of new cells

      --for new product

ORDER BY country, prod, year

/
```

The example data has no values beyond the year 2001, so any rule involving the year 2002 or later requires insertion of a new cell. The same applies to any new product name defined here. In the third rule `' 2_Products` ' is defined as a product with sales in `2005` which equal the sum of `Bounce` in `2001` and `Y Box` in `2000` .

The first rule, for `Bounce` in `2002` , inserts new cells since it is positional notation. The second rule, for `Y Box` , uses symbolic notation, but since there are already values for `' Y Box` ' in the year `2001` , it updates those values. The third rule, for `' 2_Products` ' in `2005` , is positional, so it can insert new cells, and you will see them in the output.


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

```
-------------------- --------------- ---------- ----------
Italy                Bounce                2005         20

SQL> @sym_cell1

COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce                2000         10
Italy                Bounce                2001         10

SQL> @pos_sym

COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Z_Products            2005   34169.19
Italy                Bounce                2002     4846.3
Italy                Y Box                 2001   15215.16
Japan                Z_Products            2005   51994.26
Japan                Bounce                2002     6303.6
Japan                Y Box                 2001   22161.91

6 rows selected.

SQL>
```

Multi-Cell References on the Right Side of a Rule

Back to List of Topics

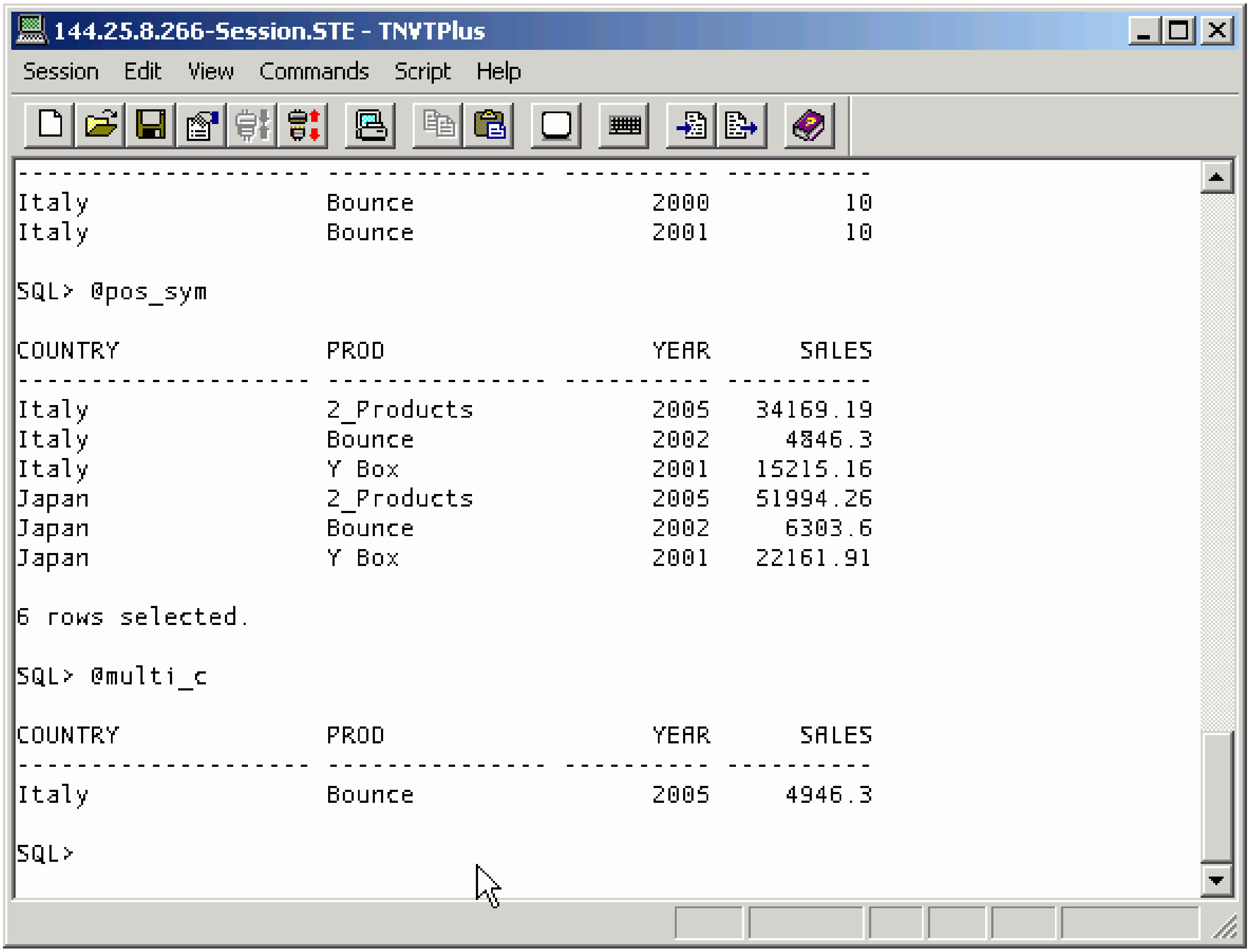
The earlier examples had multi-cell references only on the left side of the rules. If you want to refer to multiple cells on the right side of a rule, you can use multi-cell references on the right side of rules in which case an aggregate function needs to be applied on them to convert them to a single value. All existing aggregate functions including OLAP aggregates (inverse distribution functions, hypothetical rank and distribution functions etc.) and statistical aggregates, and user-defined aggregate functions can be used.

☐ **1.** You want to forecast the sales of `Bounce` in `Italy` for the year `2005` to be `100` more than the maximum sales in the period `1999` to `2001` . To do so, you need to use the `BETWEEN` clause to specify multiple cells on the right side of the rule, and these are aggregated to a single value with the `MAX()` function. From your SQL*Plus session, execute the following script:

```
@multi_c
```

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

```
SELECT SUBSTR(country,1,20) country,
       SUBSTR(prod,1,15) prod, year, sales
FROM   sales_view
WHERE  country='Italy'
  MODEL   RETURN UPDATED ROWS
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale sales)
    RULES (
      sales['Bounce', 2005] =
       100 + max(sales)['Bounce', year BETWEEN 1998 AND 2002]  )
ORDER BY country, prod, year
/
```


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

```
-------------------- --------------- ---------- ----------
Italy                Bounce                2000         10
Italy                Bounce                2001         10

SQL> @pos_sym

COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Z_Products            2005   34169.19
Italy                Bounce                2002     4846.3
Italy                Y Box                 2001   15215.16
Japan                Z_Products            2005   51994.26
Japan                Bounce                2002     6303.6
Japan                Y Box                 2001   22161.91

6 rows selected.

SQL> @multi_c

COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce                2005     4946.3

SQL>
```

Note that aggregate functions can appear only on the right side of rules. Arguments to the aggregate function can be constants, bind variables, measures of the `MODEL` clause, or expressions involving them.

The CV( ) Function and ANY Wildcard

Back to List of Topics

The `CV( )` function is a very powerful tool that makes rule creation highly productive. `CV( )` is used on the right side of rules to copy the current value of a dimension specified on the left side. It is helpful wherever the left side specifications refers to multiple cells. In terms of relational database concepts, it acts like a join operation.

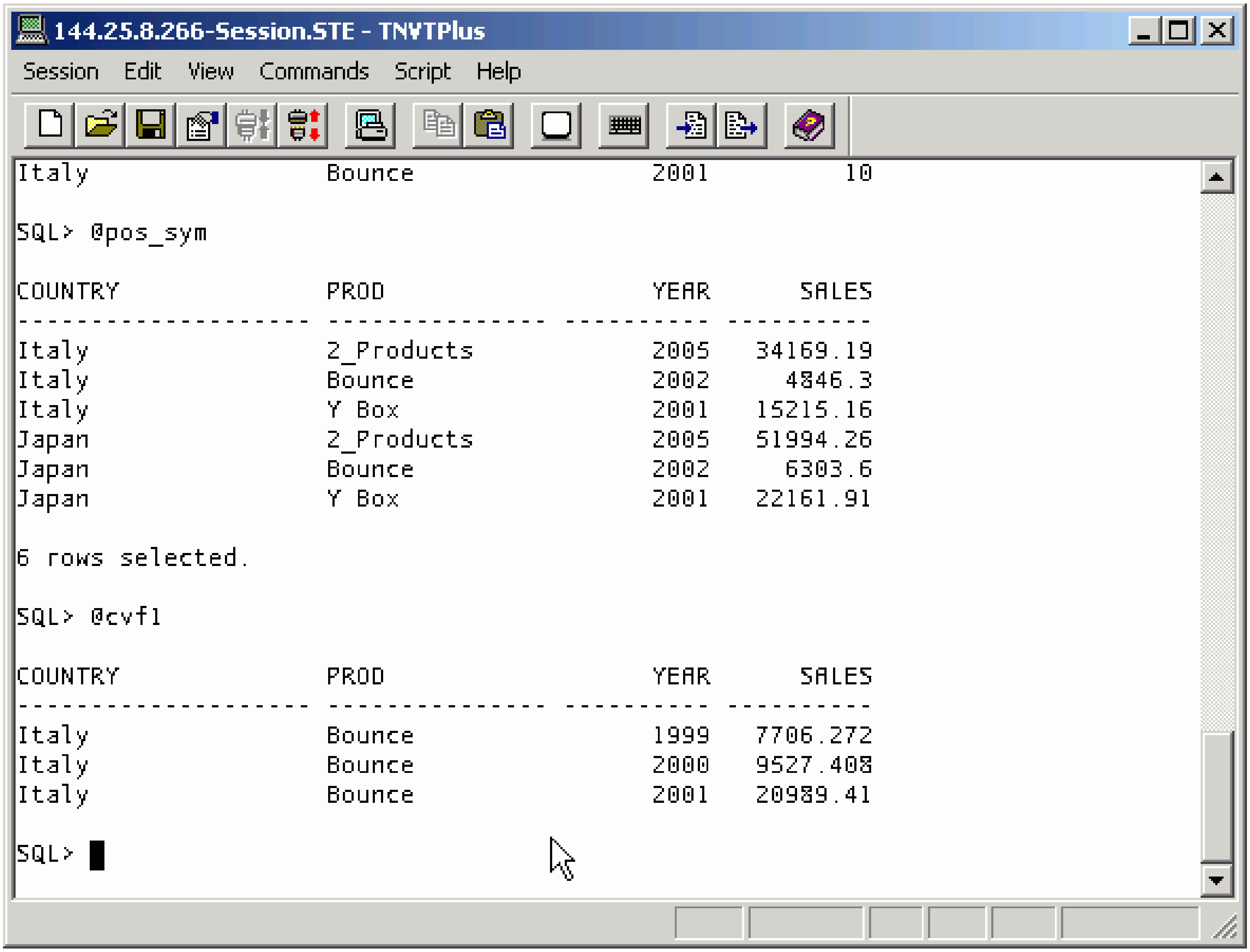
`CV( )` allows for very flexible expressions. For instance, by subtracting from the `CV(year)` value you can refer to other rows in the data set. If you have the expression ' `CV(year) -2` ' in a cell reference, you can access data from two years earlier. CV() functions are most commonly used as part of a cell reference, but they can also be used outside a cell reference as freestanding elements of an expression.

☐ **1.** You want to update the sales values for `Bounce` in `Italy` for multiple years, using a rule where each year's sales is the sum of `Mouse Pad` sales for that year plus 20% of the `' Y Box '` sales for that year. From your SQL*Plus session, execute the following script:

```
@cvf1
```

The **`cvf1.sql`** script contains the following:

```
SELECT SUBSTR(country,1,20) country,

       SUBSTR(prod,1,15) prod, year, sales

FROM   sales_view

WHERE  country='Italy'

  MODEL   RETURN UPDATED ROWS

    PARTITION BY (country)

    DIMENSION BY (prod, year)

    MEASURES (sale sales)

    RULES (

      sales['Bounce', year BETWEEN 1995 AND 2002] =

        sales['Mouse Pad', cv(year)] +

        0.2 * sales['Y Box', cv(year)])

ORDER BY country, prod, year

/
```



```
Italy                Bounce                 2001         10

SQL> @pos_sym

COUNTRY              PROD                 YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                2_Products             2005   34169.19
Italy                Bounce                 2002     4846.3
Italy                Y Box                  2001   15215.16
Japan                2_Products             2005   51994.26
Japan                Bounce                 2002     6303.6
Japan                Y Box                  2001   22161.91

6 rows selected.

SQL> @cvf1

COUNTRY              PROD                 YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce                 1999   7706.272
Italy                Bounce                 2000   9527.408
Italy                Bounce                 2001   20989.41

SQL>
```

Note that in the above results you see values for just years 1999-2001 although any year in the range 1995 to 2002 is accepted. This is because the table has data for only those years. The `CV()` function provides the current value of a `DIMENSION BY` key of the cell currently referenced on the left side. When the left side of the rule above references the cell `'Bounce'` and `1999`, the right side expression would resolve to:

```
sales['Mouse Pad', 1999] + 0.2 * sales['Y Box', 1999]
```

Similarly, when the left side references the cell `'Bounce'` and `2000`, the right side expression evaluates to:

```
sales['Mouse Pad', 2000] + 0.2 * sales['Y Box', 2000]
```

`CV()` function takes a dimension key as its argument. It is also possible to use `CV()` without any argument as in `cv()` and in which case, positional referencing is implied. The above rule can also be written as:

```
s['Bounce', year BETWEEN 1995 AND 2002] =
  s['Mouse Pad', cv()] + 0.2 * s['Y Box', cv()]
```

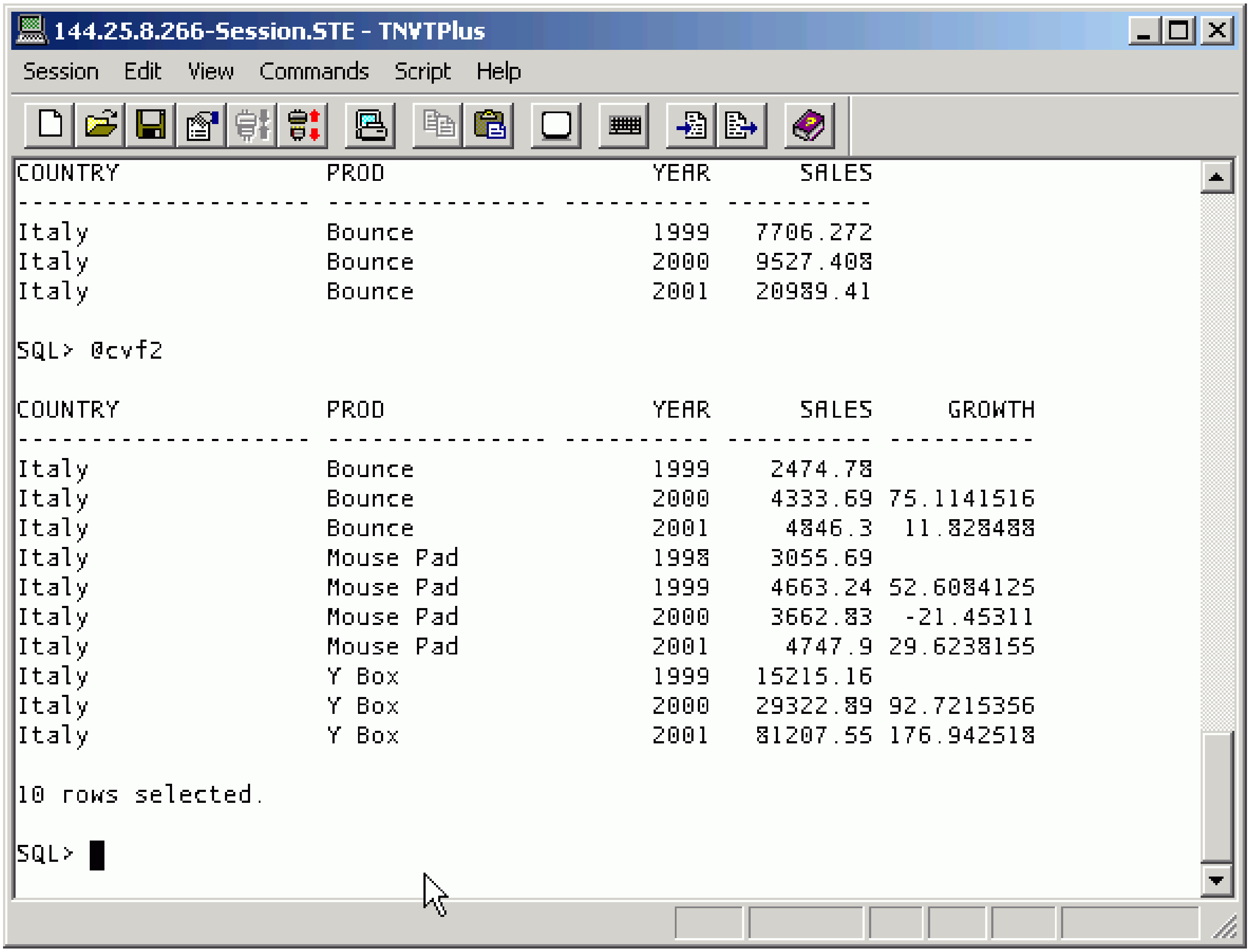
`CV()` functions can be used only in right side cell references.

☐ **2.** You want to calculate the year over year percent growth in sales for products `' Y Box'`, `' Bounce'` and `' Mouse Pad'` in `Italy`. From your SQL*Plus session, execute the following script:

```
@cvf2
```

The **`cvf2.sql`** script contains the following:

```
SELECT SUBSTR(country,1,20) country,
       SUBSTR(prod,1,15) prod, year, sales, growth
FROM   sales_view
WHERE   country='Italy'
MODEL     RETURN UPDATED ROWS
  PARTITION BY (country)
  DIMENSION BY (prod, year)
  MEASURES (sale sales, 0 growth)
  RULES  (
    growth[prod in ('Bounce','Y Box','Mouse Pad'), year between 1998 and 2001] =
      100* (sales[cv(prod), cv(year)] -
      sales[cv(prod), cv(year) -1] ) /
      sales[cv(prod), cv(year) -1] )
ORDER BY country, prod, year
/
```



```
COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Bounce                1999   7706.272
Italy                Bounce                2000   9527.408
Italy                Bounce                2001   20989.41

SQL> @cvf2

COUNTRY              PROD                  YEAR      SALES     GROWTH
-------------------- --------------- ---------- ---------- ----------
Italy                Bounce                1999    2474.78
Italy                Bounce                2000    4333.69 75.1141516
Italy                Bounce                2001     4846.3  11.828488
Italy                Mouse Pad             1998    3055.69
Italy                Mouse Pad             1999    4663.24 52.6084125
Italy                Mouse Pad             2000    3662.83  -21.45311
Italy                Mouse Pad             2001     4747.9 29.6238155
Italy                Y Box                 1999   15215.16
Italy                Y Box                 2000   29322.89 92.7215356
Italy                Y Box                 2001   81207.55 176.942518

10 rows selected.

SQL>
```

Note that the blank cells in the results are NULLs. The rule results in a null if there is no value for the product two years earlier. None of the products have a value for 1998, so in each case the 1999 growth calculation is NULL.

☐ **3.** A wild card operator is very useful for cell specification, and you can use the ANY keyword for this purpose. You can use it with the prior example to replace the specification ' `year between 1998 and 2001` ' as shown below.

ANY can be used in cell references to include all dimension values including nulls. In symbolic reference notation, use the phrase ' `IS ANY` '. Note that the ANY wildcard prevents cell insertion when used with either positional or symbolic notation.

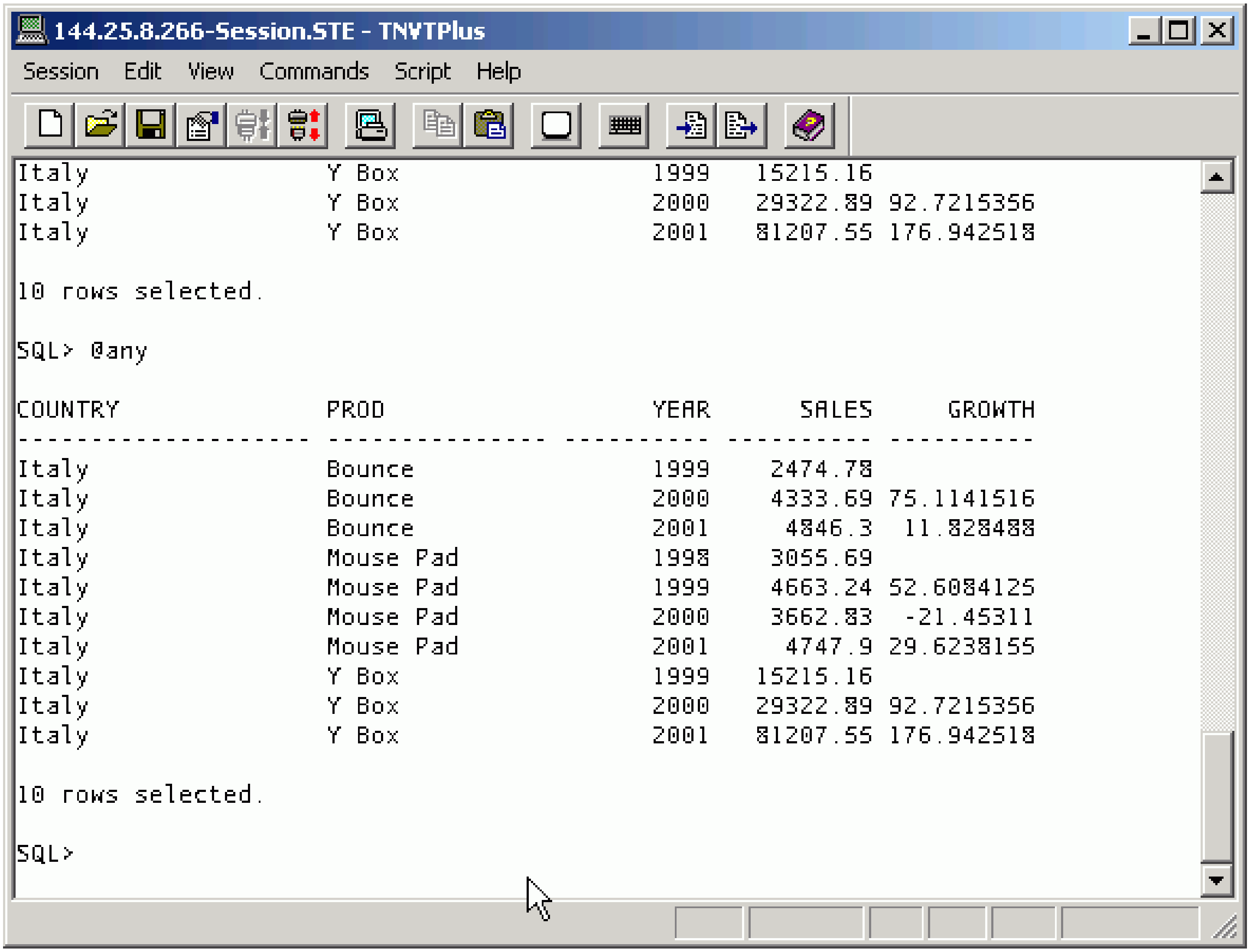
From your SQL*Plus session, execute the following script:

@any

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

```
SELECT SUBSTR(country,1,20) country,

       SUBSTR(prod,1,15) prod, year, sales, growth

FROM   sales_view

WHERE  country='Italy'

MODEL     RETURN UPDATED ROWS

  PARTITION BY (country)

  DIMENSION BY (prod, year)

  MEASURES (sale sales, 0 growth)

  RULES  (

    growth[prod in ('Bounce','Y Box','Mouse Pad'), ANY] =

       100* (sales[cv(prod), cv(year)] -

       sales[cv(prod), cv(year) -1] ) /

       sales[cv(prod), cv(year) -1] )

ORDER BY country, prod, year

/
```

This query gives the same results as the prior query because the full data set ranges from 1998 to 2001, and that is the range specified in the prior query.


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

```
Italy                Y Box                     1999   15215.16
Italy                Y Box                     2000   29322.89 92.7215356
Italy                Y Box                     2001   81207.55 176.942518

10 rows selected.

SQL> @any

COUNTRY              PROD                      YEAR      SALES     GROWTH
-------------------- --------------- ---------- ---------- ----------
Italy                Bounce                    1999    2474.78
Italy                Bounce                    2000    4333.69 75.1141516
Italy                Bounce                    2001     4846.3  11.828488
Italy                Mouse Pad                 1998    3055.69
Italy                Mouse Pad                 1999    4663.24 52.6084125
Italy                Mouse Pad                 2000    3662.83  -21.45311
Italy                Mouse Pad                 2001     4747.9 29.6238155
Italy                Y Box                     1999   15215.16
Italy                Y Box                     2000   29322.89 92.7215356
Italy                Y Box                     2001   81207.55 176.942518

10 rows selected.

SQL>
```

FOR Loops - A Concise Way to Specify New Cells

[Back to List of Topics]

The `MODEL` clause provides a `FOR` construct which can be used inside rules to express computations more concisely. The `FOR` construct is allowed on both sides of rules. For example, consider the following rules that estimate the sales of several products for year 2005 to be 30% higher than their sales for year 2001:

```
RULES
   (
   sales['Mouse Pad', 2005] = 1.3 * sales['Mouse Pad', 2001],
   sales['Bounce', 2005] = 1.3 * sales['Bounce', 2001],
   sales['Y Box', 2005] = 1.3 * sales['Y Box', 2001]
   )
```

By using positional notation on the left side of the rules, you ensure that cells for these products in the year 2005 will get inserted if they are not previously present in the array. This is rather bulky as you may have to have as many rules as there are products. If you work with dozens of products, it becomes an unwieldy approach.

You can reword this computation so it is concise and has exactly the same behavior:

```
SELECT SUBSTR(country,1,20) country,
       SUBSTR(prod,1,15) prod, year, sales
FROM sales_view
WHERE country='Italy'

MODEL RETURN UPDATED ROWS
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale sales)
    RULES (
     sales[FOR prod in ('Mouse Pad', 'Bounce', 'Y Box'), 2005] =
     1.3 * sales[cv(prod), 2001] )
ORDER BY country, prod, year;
```

This results in:

```
COUNTRY              PROD                  YEAR      SALES

-------------------- --------------- ---------- ----------

Italy                Bounce                2005   6407.245

Italy                Mouse Pad             2005    6402.63

Italy                Y Box                 2005 108308.304
```

If you write a specification similar to the above one, but without the `FOR` keyword, only cells which already exist would be updated, and no new cells would be inserted. In the SH data, that would mean no rows are returned. Here is that query:

```
SELECT SUBSTR(country,1,20) country, SUBSTR(prod,1,15) prod, year, sales
FROM sales_view
WHERE country='Italy'

MODEL RETURN UPDATED ROWS
    PARTITION BY (country)
    DIMENSION BY (prod, year)
    MEASURES (sale sales)
    RULES (
     sales[prod in ('Mouse Pad', 'Bounce', 'Y Box'), 2005] =
     1.3 * sales[cv(prod), 2001] )
ORDER BY country, prod, year;
```

```
no rows selected
```

You can view the `FOR` construct as generating multiple rules with positional references from a single rule, thus enabling creation of new cells (`UPSERT` behavior).

Note that the `MODEL` clause has a limit of 10,000 rules, and the virtual rules generated by FOR constructs are counted toward that limit. It is important to consider the total number of rules potentially generated by `FOR` constructs to avoid exceeding the rule limit.

In situations where `FOR` constructs would generate over 10,000 rules, the limit can be avoided in two ways. First, it may be possible to move dimensions into the PARTITION BY clause. This reduces the number of rules the FOR construct will generate, and the 10,000 rule maximum counts only the rules within one partition at a time. The second approach is to provide the MODEL clause a set of rows that includes all the needed cells. The `MODEL` clause then does not need to create cells, but just updates them, and this can be done without using `FOR` constructs. To generate the full set of rows needed, it is helpful to use the Partitioned Outer Join feature added in Oracle Database 10g. Partitioned Outer Join makes it easy to specify fully populated data sets. For more information, see the Oracle by Example lesson Using Partitioned Outer Join to Fill Gaps in Sparse Data .

If you know that the needed dimension values come from a sequence with regular intervals, you can use another form of the `FOR` construct:

```
FOR dimension FROM value1 TO value2  [INCREMENT | DECREMENT] value3
```

This specification results in values between `value1` and `value2` by starting from `value1` and incrementing (or decrementing) by `value3` .

☐ **1.** You want to specify projection sales values `Mouse Pad` for the years `2005` to `2012` so that they are equal to 120% of the value in `2001` . From your SQL*Plus session, execute the following script:

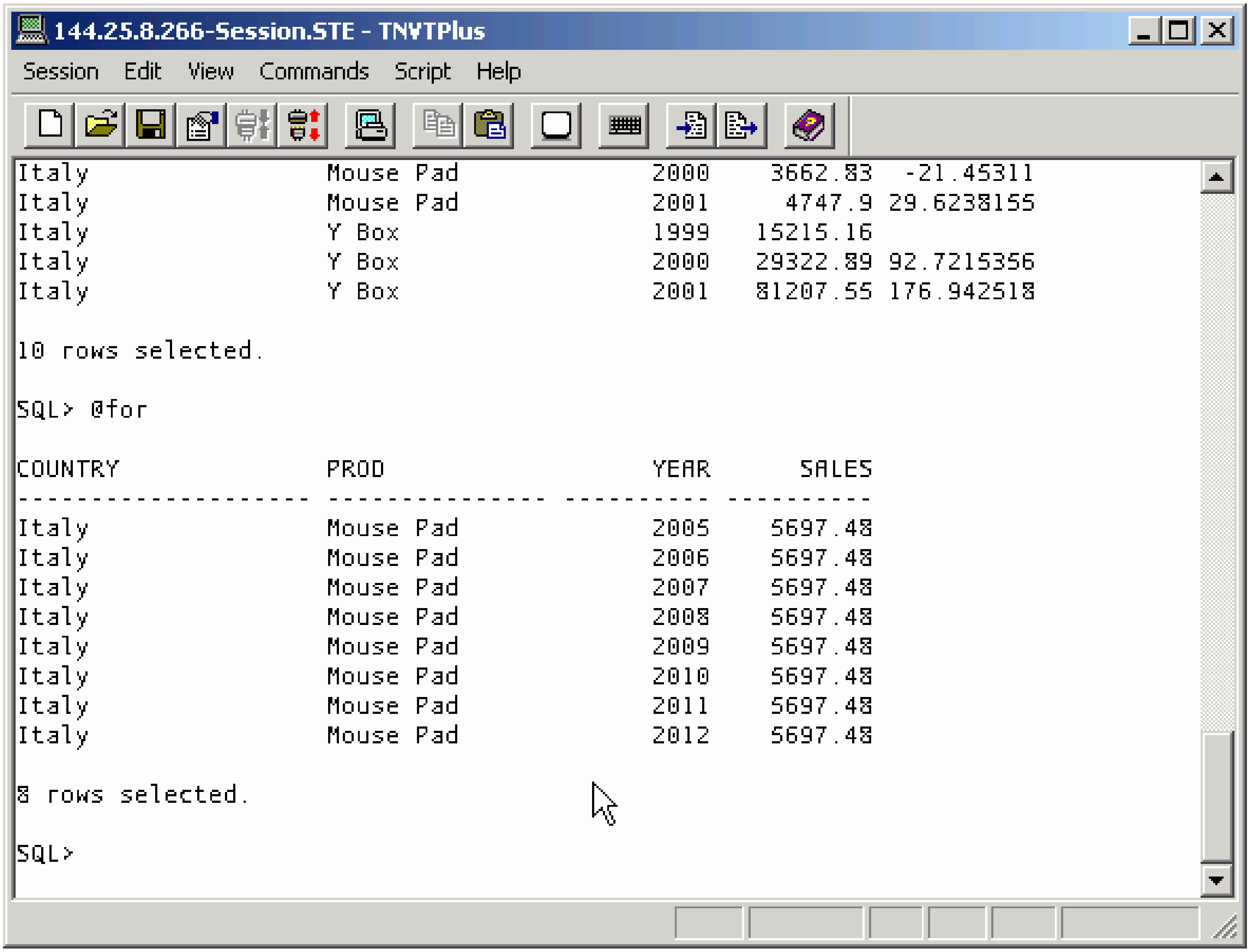
```
@for
```

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

```
SELECT SUBSTR(country,1,20) country,

       SUBSTR(prod,1,15) prod, year, sales

FROM   sales_view

WHERE  country='Italy'

MODEL     RETURN UPDATED ROWS
```

```
  PARTITION BY (country)

  DIMENSION BY (prod, year)

  MEASURES (sale sales)

  RULES  (

    sales['Mouse Pad', FOR year FROM 2005 TO 2012 INCREMENT 1] =

      1.2 * sales[cv(prod), 2001]  )

ORDER BY country, prod, year

/
```

This kind of `FOR` construct can be used for dimensions of numeric, date and datetime datatypes. The increment/decrement expression `value3` should be numeric for numeric dimensions and can be numeric or interval for dimensions of date or datetime types. There are other methods to use the `FOR` construct, and they are described in detail in the *Data Warehousing Guide* . The most important of these other methods is to use a SQL subquery as the argument for an `IN` operator. When using `FOR` constructs with subqueries, it is essential to examine the total number of rules that the `FOR` construct may generate and make sure they will not exceed the 10,000 rule limit.


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

```
Italy                Mouse Pad             2000    3662.83  -21.45311
Italy                Mouse Pad             2001     4747.9 29.6238155
Italy                Y Box                 1999   15215.16
Italy                Y Box                 2000   29322.89 92.7215356
Italy                Y Box                 2001   81207.55 176.942518

10 rows selected.

SQL> @for

COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Mouse Pad             2005    5697.48
Italy                Mouse Pad             2006    5697.48
Italy                Mouse Pad             2007    5697.48
Italy                Mouse Pad             2008    5697.48
Italy                Mouse Pad             2009    5697.48
Italy                Mouse Pad             2010    5697.48
Italy                Mouse Pad             2011    5697.48
Italy                Mouse Pad             2012    5697.48

8 rows selected.

SQL>
```

Order of Evaluation of Rules

Back to List of Topics

By default, rules are evaluated in the order they appear in the `MODEL` clause. An optional keyword ' `SEQUENTIAL ORDER` ' can be specified in the `MODEL` clause to make such an evaluation order explicit. SQL `MODEL` s with sequential rule order of evaluation are called 'Sequential Order' models.

To have models calculated so that all rule dependencies are considered and processed in correct order, use the `AUTOMATIC ORDER` keywords. When a model has a large number of rules it may be more efficient to use the `AUTOMATIC ORDER` option than to manually check that the rules are listed in a logically correct sequence. This enables more productive development and maintenance of models.

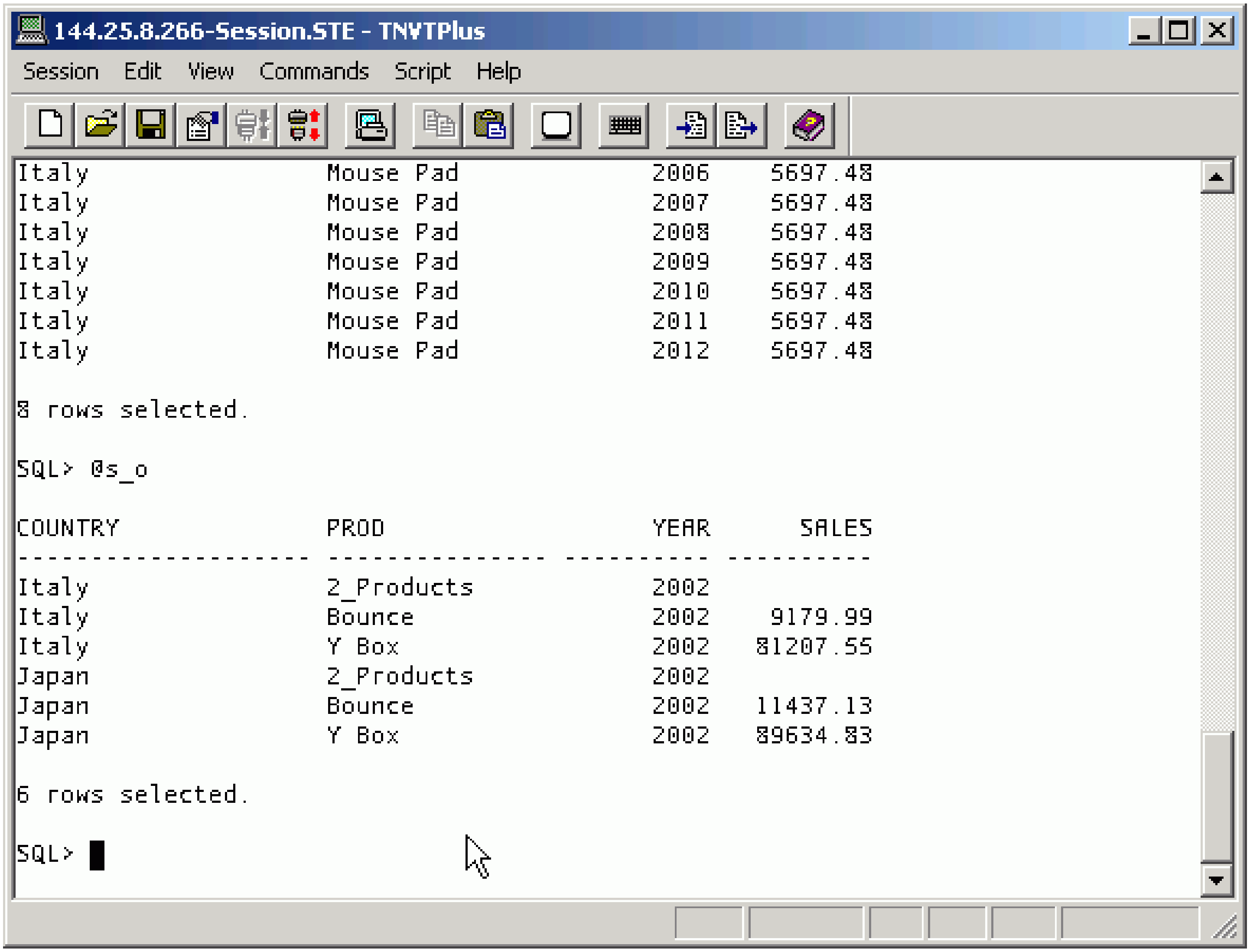
☐ **1.** You can have a model with many rules which creates new product values based on other products. To ensure that the rules will be executed in correct sequence so that no dependencies are missed use the `AUTOMATIC ORDER` keywords. The example below contains three rules to illustrate the concept. From your SQL*Plus session, execute the following script:

```
@s_o
```

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

```
SELECT SUBSTR(country,1,20) country,

       SUBSTR(prod,1,15) prod, year, sales

FROM   sales_view

WHERE  country IN  ('Italy','Japan')

  MODEL   RETURN UPDATED ROWS

    PARTITION BY (country)

    DIMENSION BY (prod, year)

    MEASURES (sale sales)

    RULES SEQUENTIAL ORDER  (

      sales['2_Products', 2002] = sales['Bounce', 2002] + sales['Y Box', 2002],

      sales['Bounce', 2002] = sales['Bounce', 2001] + sales['Bounce', 2000],

      sales['Y Box', 2002] = sales['Y Box', 2001] )

ORDER BY country, prod, year

/
```

This query returns the results for the newly created '` 2_Products `' product and calculates the values for `Bounce` and `Y Box` before `2_Products` :


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

```
Italy                Mouse Pad            2006    5697.48
Italy                Mouse Pad            2007    5697.48
Italy                Mouse Pad            2008    5697.48
Italy                Mouse Pad            2009    5697.48
Italy                Mouse Pad            2010    5697.48
Italy                Mouse Pad            2011    5697.48
Italy                Mouse Pad            2012    5697.48

8 rows selected.

SQL> @s_o

COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                2_Products            2002
Italy                Bounce                2002    9179.99
Italy                Y Box                 2002   81207.55
Japan                2_Products            2002
Japan                Bounce                2002   11437.13
Japan                Y Box                 2002   89634.83

6 rows selected.

SQL>
```

This query should not calculate the values for `Bounce` and `Y Box` before `2_Products` , and `2_Products` is assigned null values.

NULL Measures and Missing Cells

Back to List of Topics

Applications using SQL `MODEL` s would not only have to deal with non-deterministic values for a cell measure in the form of stored NULL entries, but also with non-determinism in the form of missing cells. A cell, referenced by a single cell reference, that is missing in the query's data is called a missing cell. The `MODEL` clause provides a default treatment for nulls and missing cells and also provides options that applications can use to treat non-deterministic values as per their business logic. By default, NULL cell measure values are treated the same way as nulls are treated elsewhere in SQL. Missing cells are treated as cells with NULL measure values. For example, the following query yields a NULL for sales because the dataset does not include 2004 values:

```
SELECT SUBSTR(country,1,20) country, SUBSTR(prod,1,15) prod, year, sales
```

```
FROM sales_view
WHERE country='Italy'

MODEL     RETURN UPDATED ROWS
  PARTITION BY (country)
  DIMENSION BY (prod, year)
  MEASURES (sale sales)
  RULES  (
    sales['Mouse Pad', 2005] =
    sales['Mouse Pad', 1999] + sales['Mouse Pad', 2004])
ORDER BY country, prod, year;
```

```
COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------

Italy                Mouse Pad             2005
```

Since NULL values cause many rules to return nulls, it may be more useful for you to treat nulls and missing values as 0 values. In this way, nulls will not be propagated through a set of calculations. You can use the `IGNORE NAV` option (`NAV` stands for Non-Available Values) to default nulls and missing cells to the following values:

- 0 for numeric data
- Empty string for character/string data
- 01-JAN-2001 for date type data
- NULL for all other data types

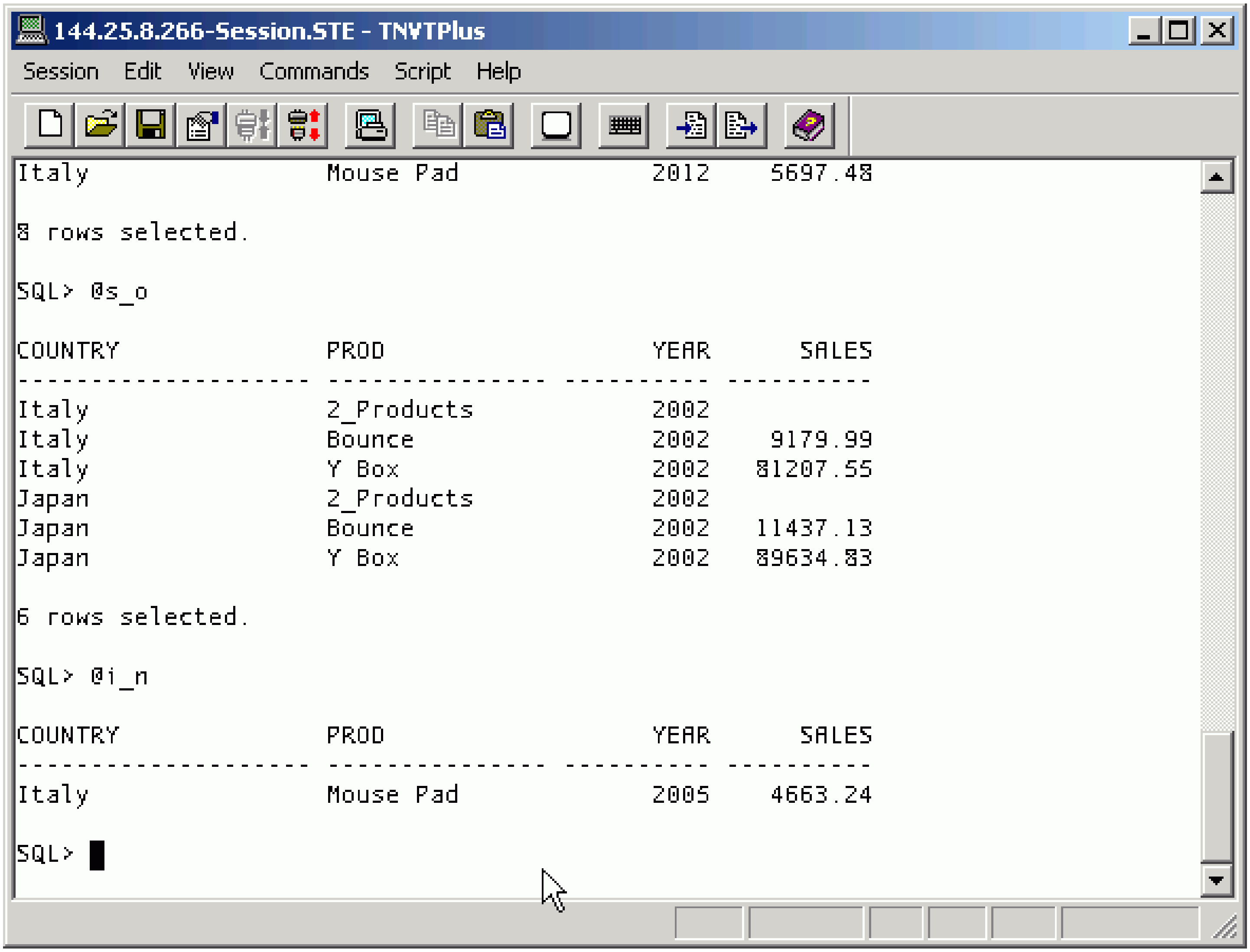
Note that the default behavior is `KEEP NAV` which treats Nulls in the standard manner and treats missing values as nulls. For more details, see the SQL `MODEL` chapter in the *Data Warehousing Guide* .

☐ **1.** Convert the query shown above to return a numeric value for sales even though the value for 2004 is missing. From your SQL*Plus session, execute the following script:

```
@i_n
```

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

```
SELECT SUBSTR(country,1,20) country,

       SUBSTR(prod,1,15) prod, year, sales

FROM   sales_view

WHERE  country='Italy'

MODEL     IGNORE NAV  RETURN UPDATED ROWS

  PARTITION BY (country)

  DIMENSION BY (prod, year)

  MEASURES (sale sales)

  RULES  (

    sales['Mouse Pad', 2005] =

    sales['Mouse Pad', 1999] + sales['Mouse Pad', 2004])

ORDER BY country, prod, year

/
```



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

```
Italy                Mouse Pad               2012    5697.48

8 rows selected.

SQL> @s_o

COUNTRY              PROD                    YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                2_Products              2002
Italy                Bounce                  2002    9179.99
Italy                Y Box                   2002   81207.55
Japan                2_Products              2002
Japan                Bounce                  2002   11437.13
Japan                Y Box                   2002   89634.83

6 rows selected.

SQL> @i_n

COUNTRY              PROD                    YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Mouse Pad               2005    4663.24

SQL>
```

Reference `MODEL` s

Back to List of Topics

In addition to the multidimensional array on which rules operate, which is called the Main SQL `MODEL` , one or more read-only multidimensional arrays, called Reference `MODEL` s, can be created and referenced in the `MODEL` clause to act as look-up tables. Using Reference `MODEL` s, you can relate objects of different dimensionality. Like a Main SQL `MODEL` , a Reference `MODEL` is defined over a query block and has the `DIMENSION BY` and `MEASURE` clauses to indicate its dimensions and measures respectively. A Reference `MODEL` is created by the following subclause of the `MODEL` clause:

```
REFERENCE model_name ON (query) DIMENSION BY (cols) MEASURES (cols) [reference    options]
```

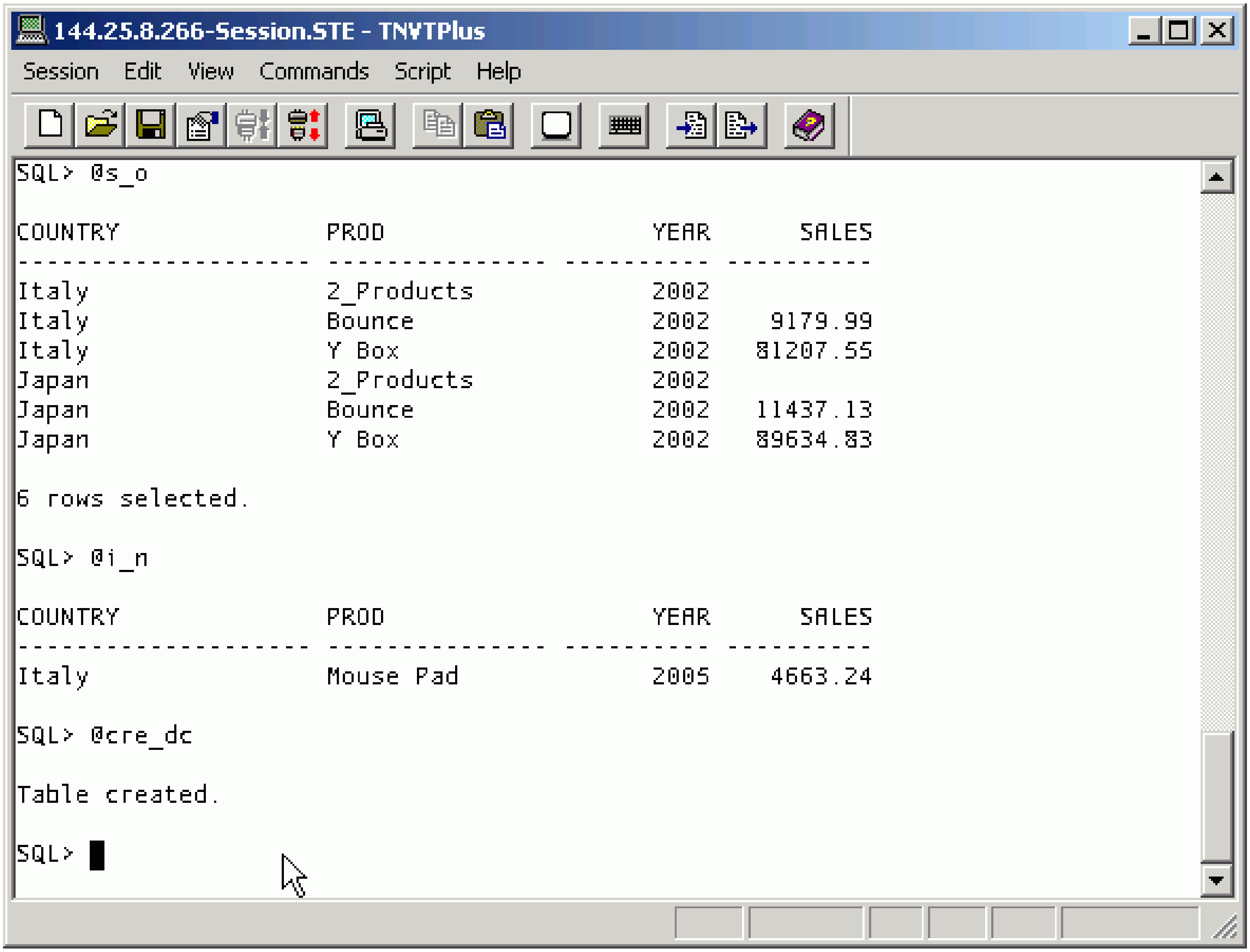
Reference models can be used only in the right side of rules and the `PARTITION` clause is not available in reference models.

☐ **1.** Convert projected sales figures of different countries, each in their own currency, into US currency and show both figures. You need to create a table with conversion ratios of local currencies to the US dollar. From your SQL*Plus session, execute the following script:

```
@cre_dc
```

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

```
CREATE TABLE dollar_conv(country VARCHAR2(30), exchange_rate NUMBER)

/
```



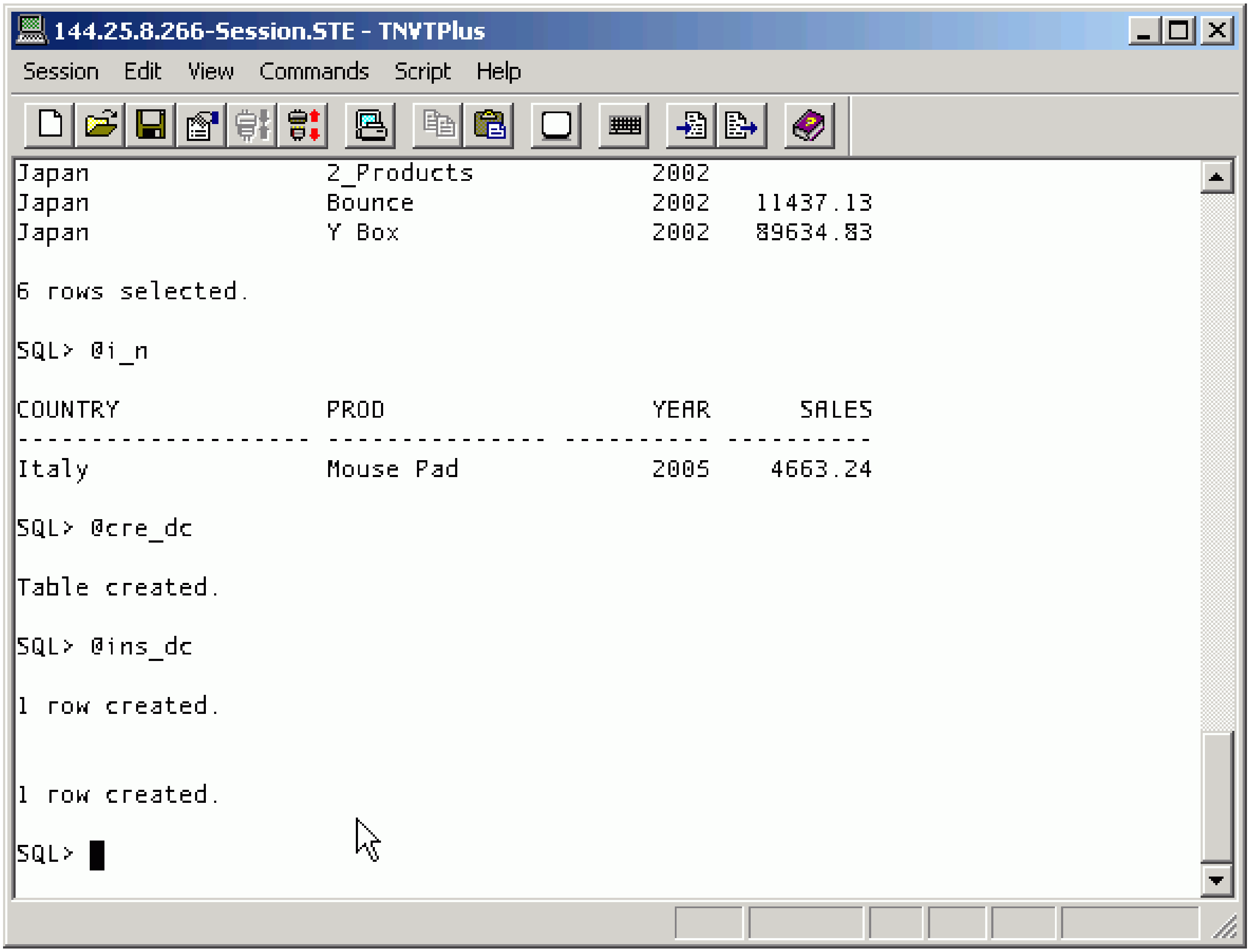
```
SQL> @s_o

COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                2_Products            2002
Italy                Bounce                2002    9179.99
Italy                Y Box                 2002   81207.55
Japan                2_Products            2002
Japan                Bounce                2002   11437.13
Japan                Y Box                 2002   89634.83

6 rows selected.

SQL> @i_n

COUNTRY              PROD                  YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                Mouse Pad             2005    4663.24

SQL> @cre_dc

Table created.

SQL>
```

☐ **2.** Insert two rows into the `DOLLAR_CONV` table. From your SQL*Plus session, execute the following script:

```
@ins_dc
```

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

```
INSERT INTO dollar_conv VALUES('Canada', 0.75)

/

INSERT INTO dollar_conv VALUES('Brazil', 0.14)

/
```



```
Japan                    2_Products             2002
Japan                    Bounce                 2002    11437.13
Japan                    Y Box                  2002    89634.83

6 rows selected.

SQL> @i_n

COUNTRY                  PROD                   YEAR       SALES
-------------------- --------------- ---------- ----------
Italy                    Mouse Pad              2005     4663.24

SQL> @cre_dc

Table created.

SQL> @ins_dc

1 row created.


1 row created.

SQL>
```

☐ **3.** Base the sales on the 2001 figures and project market growth by 2005 to be 22% in Canada and 34% in Brazil.

To convert the projected sales of Canada and Brazil for year 2005 to US dollars, you can use a Reference `MODEL` . From your SQL*Plus session, execute the following script:

```
@rm
```

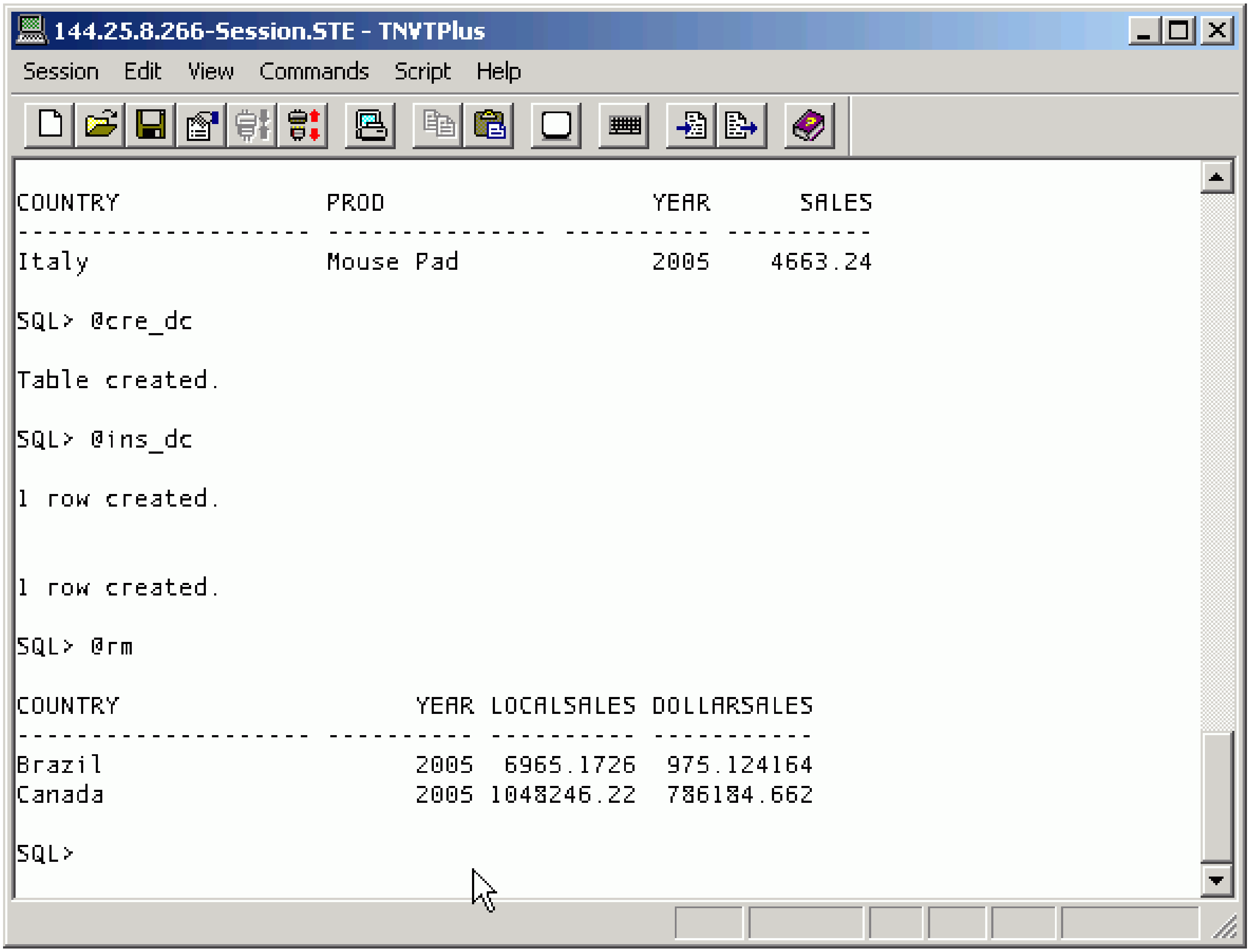
The **`rm.sql`** script contains the following:

```
SELECT   SUBSTR(country,1,20) country, year, localsales, dollarsales

FROM     sales_view

WHERE    country IN ( 'Canada', 'Brazil')

GROUP BY country, year

MODEL  RETURN UPDATED ROWS

  REFERENCE conv_refmodel ON (

    SELECT country, exchange_rate AS er FROM dollar_conv)

    DIMENSION BY (country) MEASURES (er) IGNORE NAV

  MAIN  main_model

    DIMENSION BY (country, year)

    MEASURES (SUM(sale) sales, 0  localsales, 0 dollarsales) IGNORE NAV

    RULES  (

      /* assuming that sales in  Canada grow by 22% */

      localsales['Canada', 2005] = sales[cv(country), 2001] * 1.22,

        dollarsales['Canada', 2005] = sales[cv(country), 2001] * 1.22 *

        conv_refmodel.er['Canada'],

      /* assuming that economy in Brazil grows by 34% */

      localsales['Brazil', 2005] = sales[cv(country), 2001] * 1.34,

      dollarsales['Brazil', 2005] = sales['Brazil', 2001] * 1.34 * er['Brazil']

    )
```

```
/
```

Note the following:

- A one dimensional reference model named `CONV_REFMODEL` is created on rows from the `DOLLAR_CONV` table and that its measure `EXCHANGE_RATE` named `ER` has been referenced in the rules of the main model.
- The main model has the optional keyword `MAIN` at the start of its specification, giving it the alias ' `MAIN_MODEL` '. The keyword `MAIN` makes it easier to note the start of the main model specification. `MAIN_MODEL` has two dimensions, `COUNTRY` and `YEAR` , whereas the reference model `DOLLAR_CONV` has one dimension `country` .
- You can use different styles of accessing the `EXCHANGE_RATE` measure of the reference model: for Canada it is explicit with model_name.measure_name notation `CONV_REFMODEL.ER` whereas for Brazil, it is a simple measure_name reference ' `ER` '. The former notation needs to be used to resolve any ambiguities in column names across main and reference models.
- Use the placeholder value of `0` when specifying the new measures `LOCALSALES` and `DOLLARSALES` . Other numbers would also work as placeholder value



```
COUNTRY                    PROD                      YEAR      SALES
-------------------- --------------- ---------- ----------
Italy                PROD Mouse Pad            2005    4663.24

SQL> @cre_dc

Table created.

SQL> @ins_dc

1 row created.


1 row created.

SQL> @rm

COUNTRY                    YEAR LOCALSALES DOLLARSALES
-------------------- ---------- ---------- -----------
Brazil                     2005  6965.1726  975.124164
Canada                     2005 1048246.22  786184.662

SQL>
```

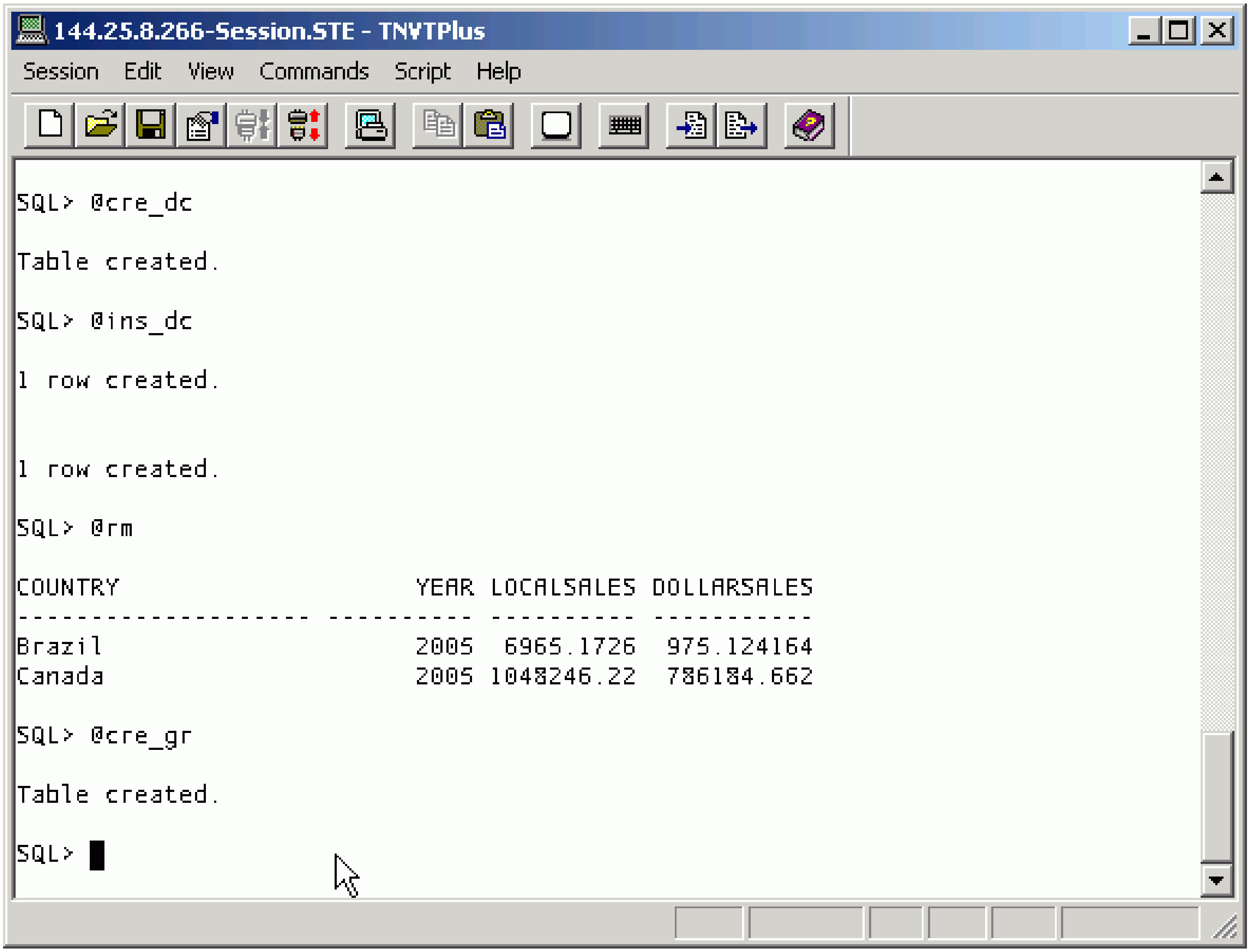
Growth rates in this example are hard coded in the rules: growth rate for Canada is 22% and that of Brazil is 34%. Your rules would be much more flexible if they could work with growth values looked up from a separate table of growth rates. Such a table could cover many years and countries.

☐ **4.** Use both exchange rate and growth rate reference models to find the projected sales in local currency and U.S. dollars for 2002. Create a table that stores the percentage growth by country and year. From your SQL*Plus session, execute the following script:

```
@cre_gr
```

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

```
CREATE TABLE growth_rate(country     VARCHAR2(30),
                         year        NUMBER,
                         growth_rate NUMBER)
/
```


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

```
SQL> @cre_dc

Table created.

SQL> @ins_dc

1 row created.


1 row created.

SQL> @rm

COUNTRY                    YEAR LOCALSALES DOLLARSALES
-------------------- ---------- ---------- -----------
Brazil                     2005  6965.1726  975.124164
Canada                     2005 1048246.22  786184.662

SQL> @cre_gr

Table created.

SQL>
```

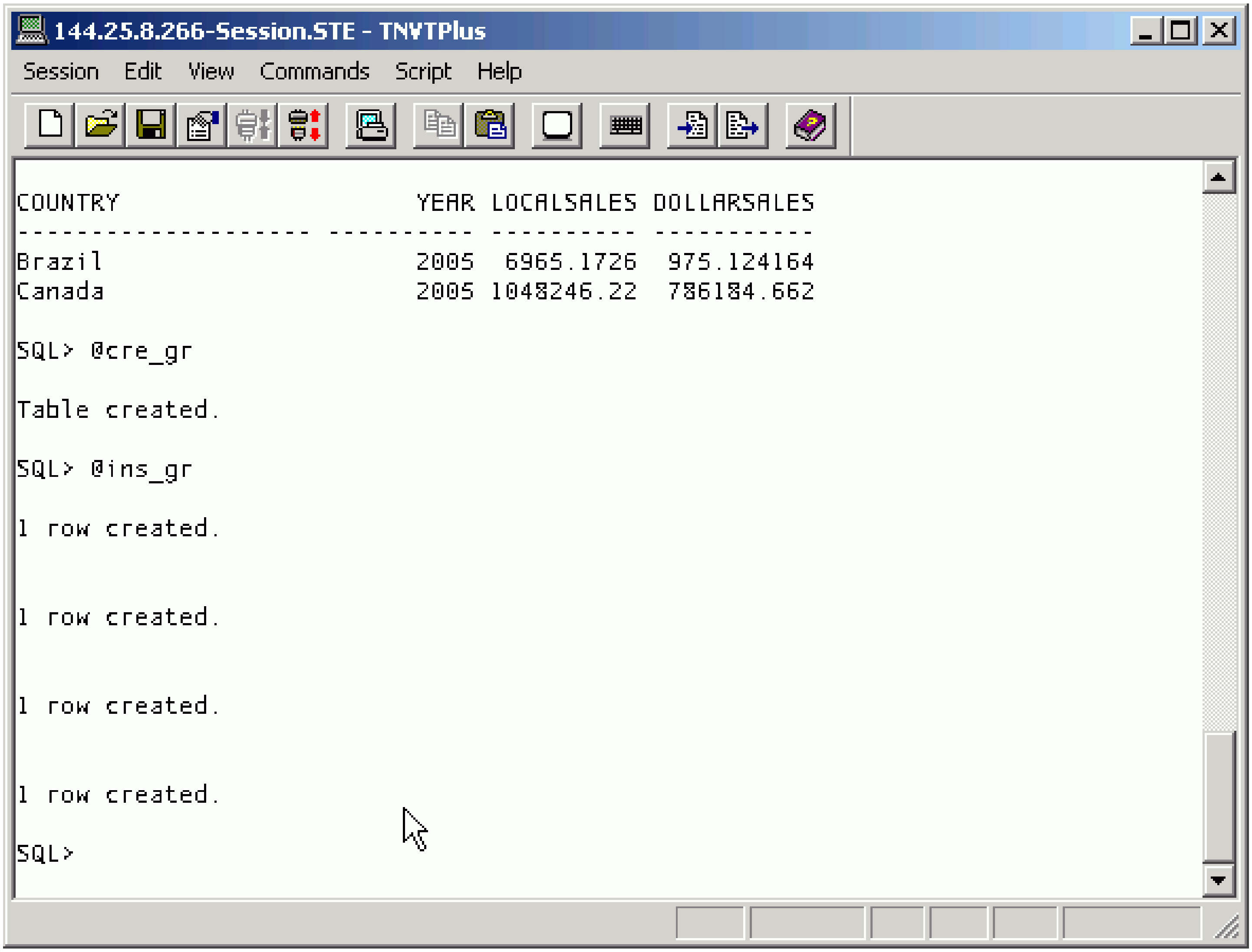
☐ **5.** Insert rows into the `GROWTH_RATE` table. From your SQL*Plus session, execute the following script:

```
@ins_gr
```

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

```
INSERT INTO growth_rate VALUES('Brazil', 2002, 2.5)

/

INSERT INTO growth_rate VALUES('Brazil', 2003, 5)

/
```

```
INSERT INTO growth_rate VALUES('Canada', 2002, 3)

/

INSERT INTO growth_rate VALUES('Canada', 2003, 2.5)

/
```



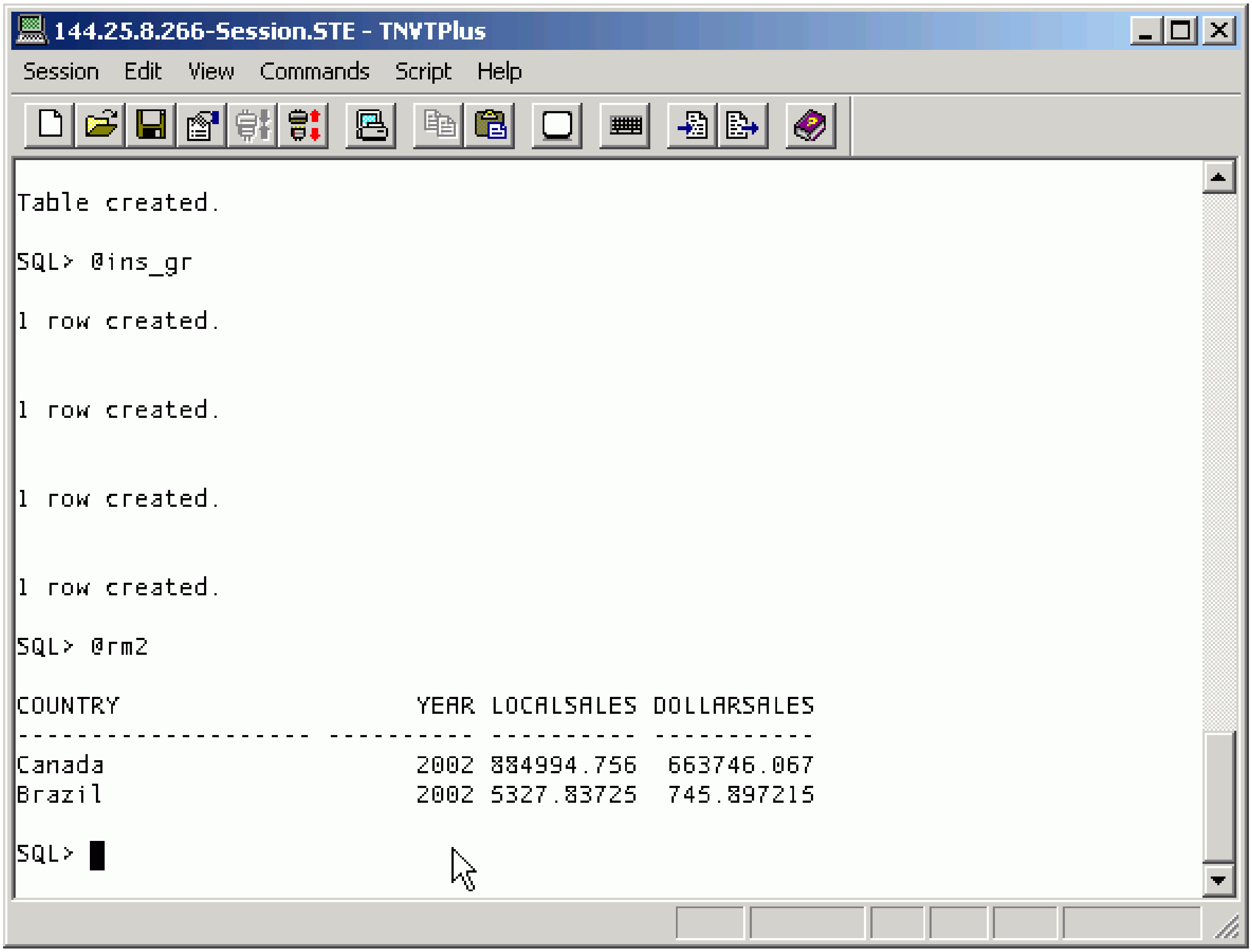
```
COUNTRY                    YEAR LOCALSALES DOLLARSALES
-------------------- ---------- ---------- -----------
Brazil                     2005  6965.1726  975.124164
Canada                     2005 1048246.22  786184.662

SQL> @cre_gr

Table created.

SQL> @ins_gr

1 row created.


1 row created.


1 row created.


1 row created.

SQL>
```

☐ **6.** Write a query that calculates sales for Brazil and Canada, applying the 2002 growth figures and converting the values to dollars. Use the reference model shown below in your query. From your SQL*Plus session, execute the following script:

```
@rm2
```

The **`rm2.sql`** script contains the following:

```
SELECT   SUBSTR(country,1,20) country, year, localsales, dollarsales

FROM     sales_view

WHERE    country IN  ('Canada','Brazil')

GROUP BY country, year

MODEL  RETURN UPDATED ROWS

  REFERENCE conv_refmodel  ON  (

    SELECT country, exchange_rate FROM dollar_conv)

    DIMENSION BY (country c)

    MEASURES (exchange_rate er) IGNORE NAV

  REFERENCE growth_refmodel ON  (

    SELECT country, year, growth_rate FROM growth_rate)

    DIMENSION BY (country c, year y)

    MEASURES (growth_rate gr) IGNORE NAV

  MAIN  main_model

    DIMENSION BY (country, year)

    MEASURES (SUM(sale) sales, 0 localsales, 0  dollarsales) IGNORE NAV

    RULES  (

      localsales[FOR country IN ('Brazil', 'Canada'), 2002] =

        sales[cv(country), 2001] *

        (100 + gr[cv(country), cv(year)])/100  ,
      dollarsales[FOR country IN ('Brazil', 'Canada'),2002] =
        sales[cv(country), 2001] *
        (100 + gr[cv(country), cv(year)])/100  *
        er[cv(country)]
    )
/
```



```
Table created.

SQL> @ins_gr

1 row created.


1 row created.


1 row created.


1 row created.

SQL> @rm2

COUNTRY                    YEAR LOCALSALES DOLLARSALES
-------------------- ---------- ---------- -----------
Canada                     2002 884994.756  663746.067
Brazil                     2002 5327.83725  745.897215

SQL>
```

Note the following:

- This query shows the capability of the `MODEL` clause in dealing with objects of different dimensionality. The Reference model `CONV_REFMODEL` has one dimension whereas the Reference `MODEL` `GROWTH_REFMODEL` and the Main SQL `MODEL` have two dimensions.

- Dimensions in the single cell references on Reference `MODEL` s are specified using the `CV()` function, thus relating the cells in Main SQL `MODEL` with the Reference `MODEL` . This specification, in effect, is performing a relational join between Main and Reference `MODEL` s.

- By using the `FOR` construct, each rule can work with multiple countries, reducing the amount of coding.
- If you added the `FOR` construct to the `YEAR` dimension on the left side of the rules and `CV(year)` expressions to the right side, you could generalize the rule to multiple years.

Iterative `MODEL` s

Back to List of Topics

Using `ITERATE` option of the `MODEL` clause, you can evaluate rules iteratively a specified number of times. The number of iterations is specified as an argument to the `ITERATE` clause. `ITERATE` can be specified only for `SEQUENTIAL ORDER` models. Use iterative models to calculate models where the rules are interdependent.

The syntax of the `ITERATE` clause is:

```
ITERATE (number_of_iterations) [ UNTIL (condition) ]
```

The `number_of_iterations` argument to `ITERATE` clause is a positive integer constant. Optionally, you can specify an early termination condition to stop rule evaluation before reaching the maximum iteration. This condition is specified in the `UNTIL` subclause of `ITERATE` and is checked at the end of an iteration. So, you will always have at least one iteration when `ITERATE` is specified.

Iterative evaluation will stop either after finishing the specified number of iterations or when the termination condition evaluates to TRUE, whichever comes first. In some cases you may want the termination condition to be based on the change, across iterations, in value of a cell. Oracle Database 10 *g* provides a mechanism to specify such conditions by allowing you to access cell values as they existed before and after the current iteration in the `UNTIL` condition. Use the `PREVIOUS` function which takes a single cell reference as argument and returns the measure value of the cell as it existed after the previous iteration. You can also access the current iteration number by using the system variable `ITERATION_NUMBER` . `ITERATION_NUMBER` starts at value `0` and is incremented after each iteration. By using `PREVIOUS` and `ITERATION_NUMBER` , you can construct complex termination conditions.

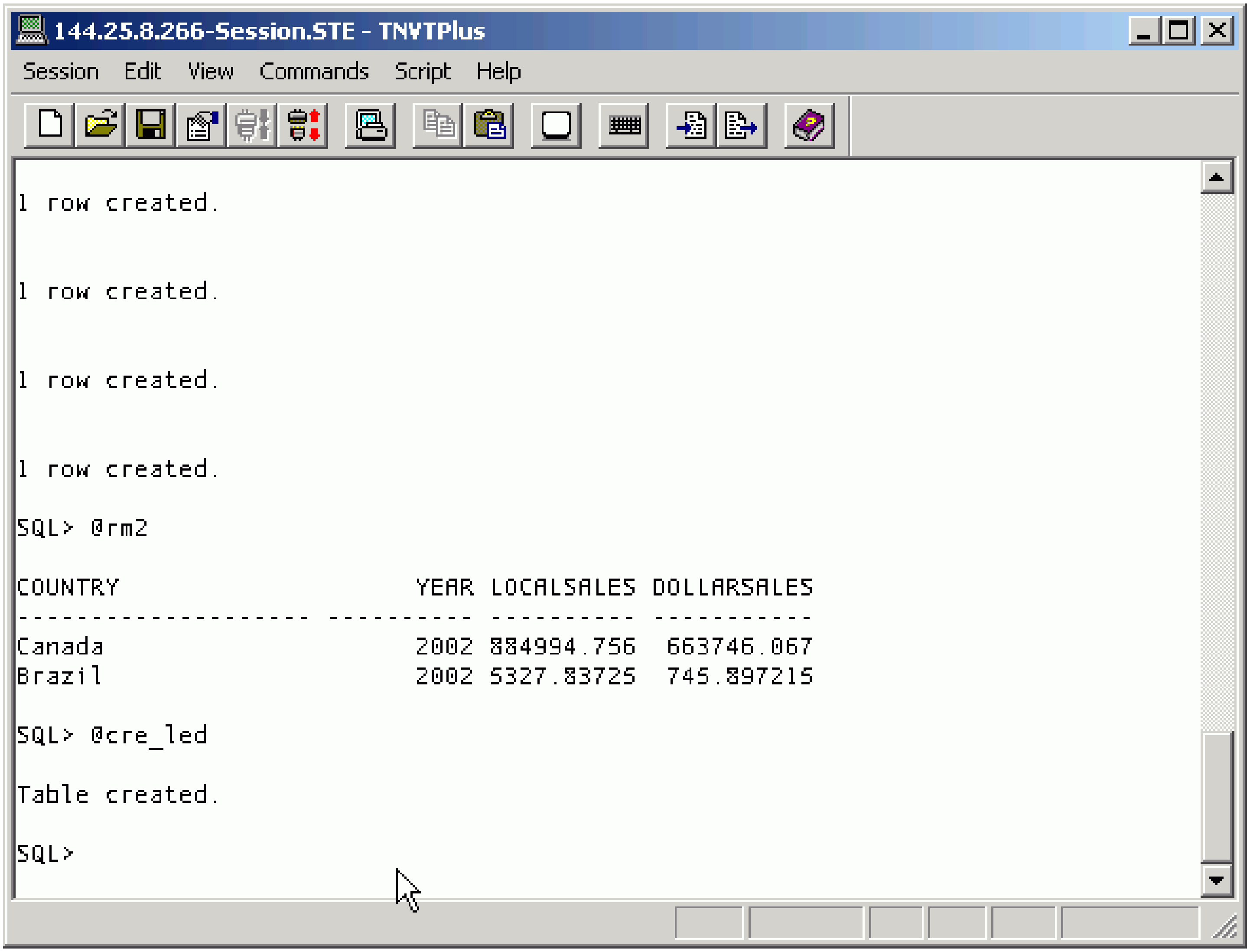
☐ **1.** You want to do financial planning for a person who earns a salary of $100,000 and has a capital gain of $15,000. His net income will be calculated as salary minus interest payments minus taxes. He pays tax-deductible interest on a loan. He also pays taxes at two rates: 28% for the salary income after interest expense is deducted, and 38% on capital gains. This person would like his interest expense to represent exactly 30% of his income. How can you calculate the taxes, interest expense and net income that will result?

All values of this scenario are stored in a table called `LEDGER` . The table holds the labels for a financial item in one column and the value of the item in another. From your SQL*Plus session, execute the following script:

```
@cre_led
```

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

```
CREATE TABLE  ledger  (account  VARCHAR2(20), balance  NUMBER(10,2) )
/
```



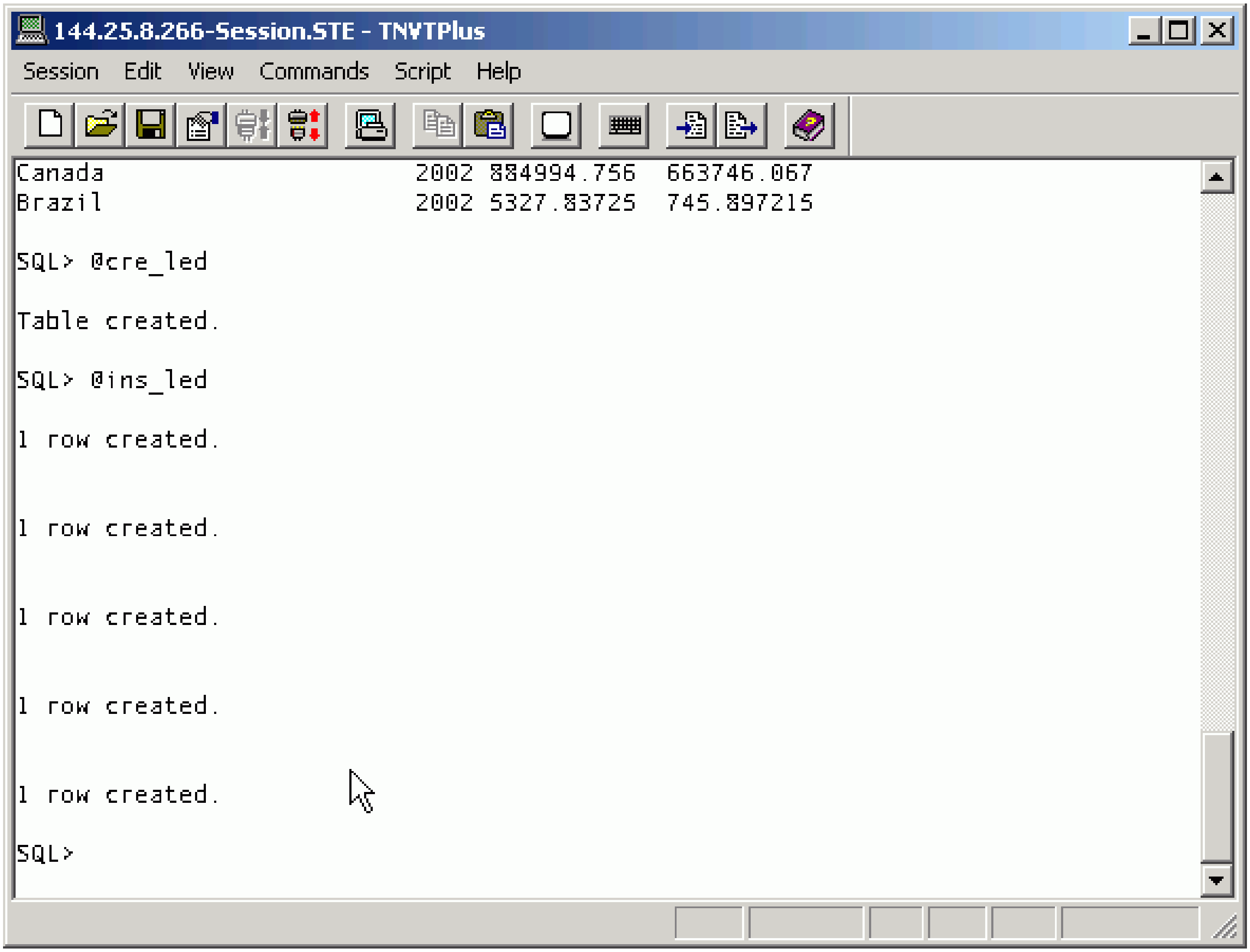
```
1 row created.


1 row created.


1 row created.


1 row created.

SQL> @rm2

COUNTRY                    YEAR LOCALSALES DOLLARSALES
-------------------- ---------- ---------- -----------
Canada                     2002 884994.756  663746.067
Brazil                     2002 5327.83725  745.897215

SQL> @cre_led

Table created.

SQL>
```

☐ **2.** Insert rows into the `LEDGER` table. From your SQL*Plus session, execute the following script:

```
@ins_led
```

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

```
INSERT INTO ledger VALUES  ('Salary', 100000)

/

INSERT INTO ledger VALUES  ('Capital_gains', 15000)

/

INSERT INTO ledger VALUES  ('Net', 0)

/

INSERT INTO ledger VALUES  ('Tax', 0)

/

INSERT INTO ledger VALUES  ('Interest', 0)

/
```


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

```
Canada                        2002 884994.756  663746.067
Brazil                        2002 5327.83725  745.897215

SQL> @cre_led

Table created.

SQL> @ins_led

1 row created.


1 row created.


1 row created.


1 row created.


1 row created.

SQL>
```

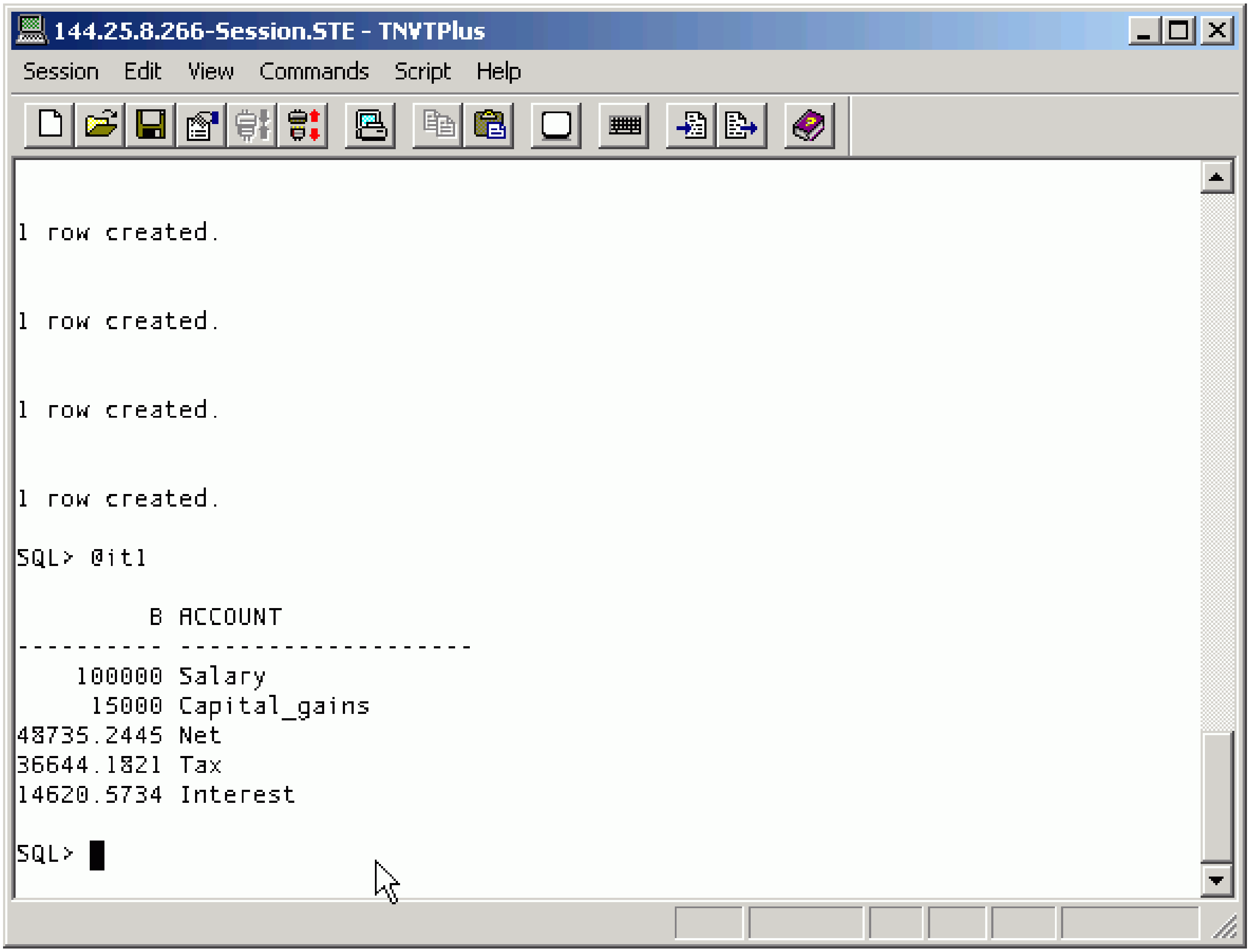
☐ **3.** To perform the calculations, use the `ITERATE` option to have the calculations repeated as many times as desired. The first pass will insert the values stored in the `LEDGER` table into the right side of the rules and create a new set of values for `NET` , `TAX` and `INTEREST` . The second pass will calculate a new set of values for `NET` , `TAX` , and `INTEREST` using the `TAX` and `INTEREST` values calculated in the prior pass. This cycle will be repeated a total of 100 times. From your SQL*Plus session, execute the following script:

```
@it1
```

The **`it1.sql`** script contains the following:

```
SELECT b, account

FROM ledger
```

```
    MODEL  IGNORE NAV
      DIMENSION BY (account)
      MEASURES (balance b)
      RULES ITERATE (100)  (
        b['Net'] =  b['Salary'] - b['Interest'] - b['Tax'],
        b['Tax'] = (b['Salary'] - b['Interest']) * 0.38 +
                    b['Capital_gains'] *0.28,
        b['Interest'] = b['Net'] * 0.30
    )
/
```



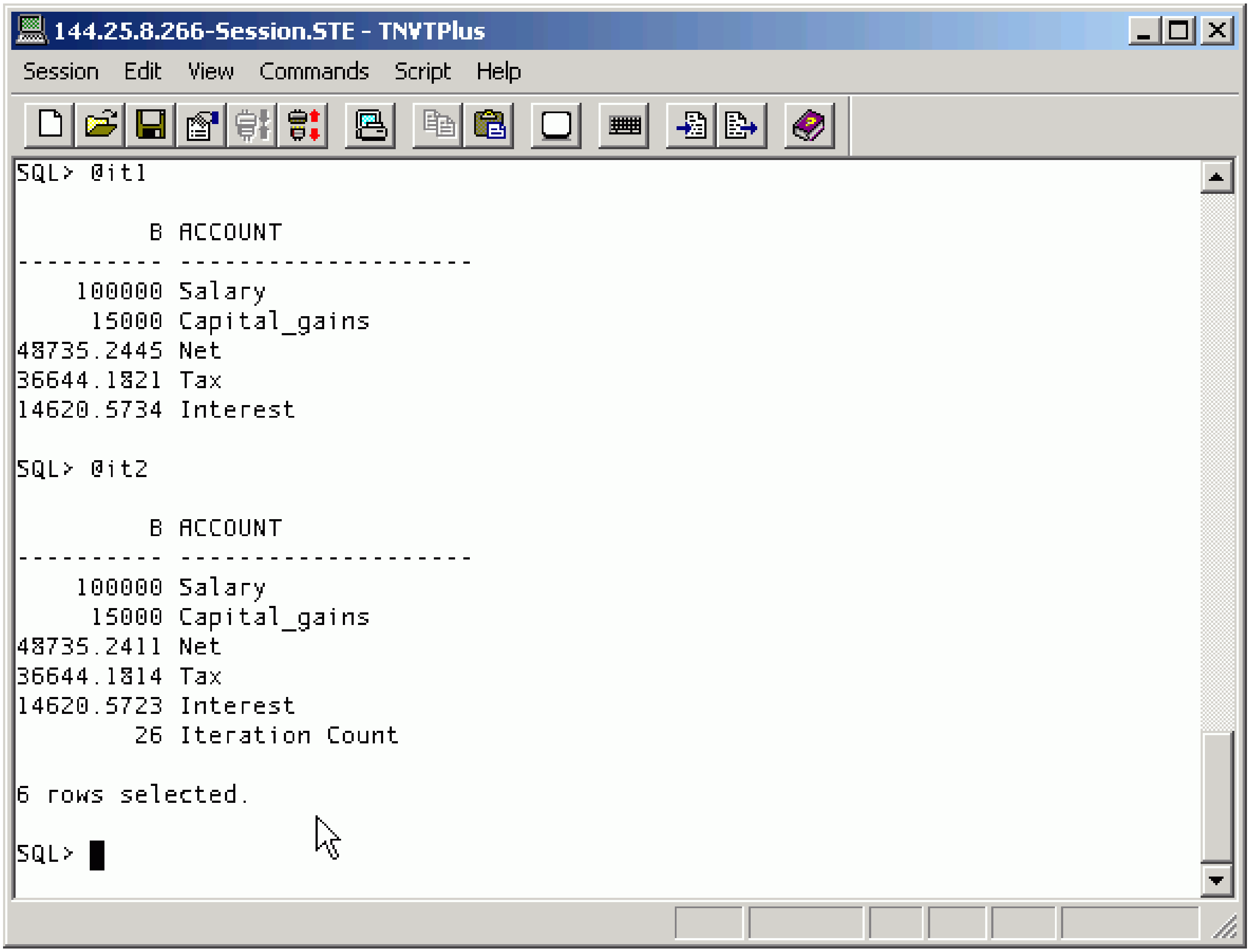
```
1 row created.


1 row created.


1 row created.


1 row created.

SQL> @it1

         B ACCOUNT
---------- --------------------
    100000 Salary
     15000 Capital_gains
48735.2445 Net
36644.1821 Tax
14620.5734 Interest

SQL>
```

☐ **4.** Write a query to avoid unnecessary processing time in the prior example. Monitor the results after each loop is complete. If the value of certain results have stopped changing by a significant amount you can stop the cycles at that point. From your SQL*Plus session, execute the following script:

```
@it2
```

The **`it2.sql`** script contains the following:

```
SELECT b, account

FROM ledger

MODEL  IGNORE NAV

  DIMENSION BY (account)

  MEASURES (balance b)

  RULES ITERATE (100)

    UNTIL ( ABS( (PREVIOUS(b['Net']) -  b['Net']) ) <  0.01 ) (

    b['Net'] = b['Salary'] - b['Interest'] - b['Tax'],

    b['Tax'] = (b['Salary'] - b['Interest']) * 0.38 +

                b['Capital_gains'] *0.28,

    b['Interest'] = b['Net'] * 0.30,

    b['Iteration Count']= ITERATION_NUMBER + 1

      -- the '+1' is needed because the ITERATION_NUMBER starts at 0

  )

/
```



```
SQL> @it1

         B ACCOUNT
---------- --------------------
    100000 Salary
     15000 Capital_gains
48735.2445 Net
36644.1821 Tax
14620.5734 Interest

SQL> @it2

         B ACCOUNT
---------- --------------------
    100000 Salary
     15000 Capital_gains
48735.2411 Net
36644.1814 Tax
14620.5723 Interest
        26 Iteration Count

6 rows selected.

SQL>
```

Note that:

- The `ABS()` function is used as part of the `UNTIL` clause. This ensures that the difference between the previous and current value can be either positive or negative as long as it is smaller than the condition.

- With the rule `s['Iteration Count']= ITERATION_NUMBER+1` , a new row called `Iteration Count` is defined. It is assigned the value of the variable `ITERATION_NUMBER` , thus tracking number of loops performed.

- In this example you see that only 26 loops were needed to get the example close to a steady state. By stopping here, an extra 74 iterations were avoided.

Ordered Rules

Back to List of Topics

An ordered rule is one that has `ORDER BY` specified on the left side. It accesses cells in the order prescribed by `ORDER BY` and applies the right side computation. This is an important issue because, when you have positional `ANY` and/or symbolic references on the left side of a rule, you might receive an error saying that the rule's results depend on the order in which cells are accessed and hence are non-deterministic. Consider the `MODEL` below:

```
SELECT year, sales
FROM   sales_view
WHERE  country='Italy' AND prod='Bounce'
  MODEL
    DIMENSION BY (year )
    MEASURES (sale sales)
    RULES SEQUENTIAL ORDER (
     sales[ANY] = sales[CV(year)-1]
     )
ORDER BY year;
```

This query returns an error message because the results are indeterminate: the values depend on the order of cell access. The query attempts to set, for all years, the sales value for a year to the sales value of the prior year. Unfortunately, the result of this rule depend on the order in which the cells are accessed. If cells are accessed in the ascending order of year, the result would be as shown in the third column of the table below. There is no 1998 value, so 1999 would have a NULL assigned to it. This NULL would be carried forward into all following assignments. If the cells are accessed in descending year order, the results would be as shown in the fourth column. 2000 has a valid value which can be assigned to 2001, and the same is true for 2000 and 2001. Therefore only 1999 is assigned a NULL (because there is no value for 1998) when access is in descending year order.

Year	**Sales**	**Current Yr if ascending**	**Prior Yr Sales if descending**
1999	2472.13	NULL	NULL
2000	4370.43	NULL	2472.13
2001		NULL	4370.43

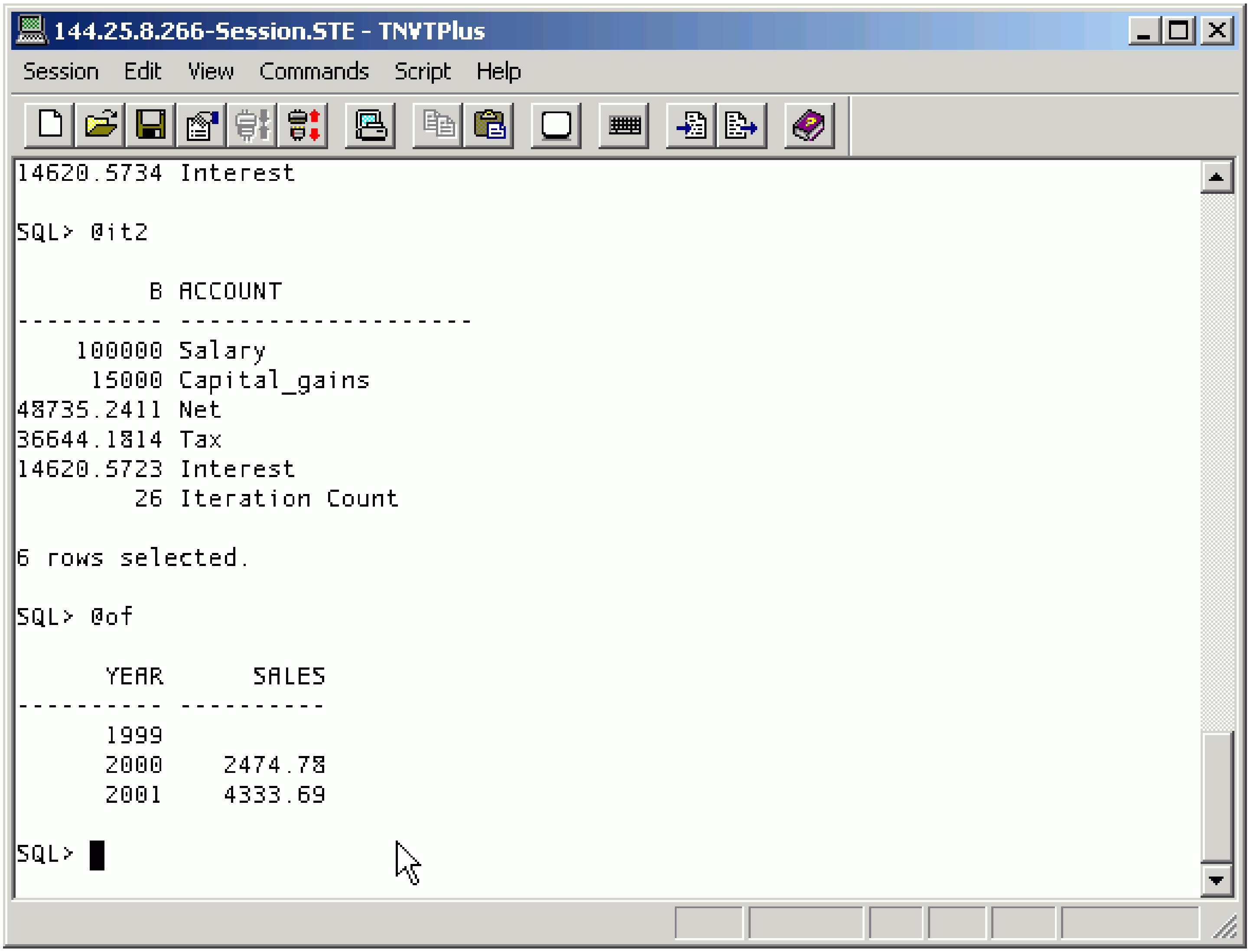
☐ **1.** Based on the above information, write a query that ensures the cells in this query are accessed in descending year order and returns non-NULL results. You will need to add the `ORDER BY` clause to the rule. From your SQL*Plus session, execute the following script:

```
@of
```

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

```
SELECT year, sales

FROM sales_view

WHERE country='Italy' AND prod='Bounce'

MODEL
```

```
  DIMENSION BY (year )

  MEASURES (  sale  sales)

  RULES SEQUENTIAL ORDER  (

    sales[ANY] ORDER BY year DESC= sales[cv(year)-1]

  )

ORDER BY year

/
```



```
14620.5734 Interest

SQL> @it2

         B ACCOUNT
---------- --------------------
    100000 Salary
     15000 Capital_gains
48735.2411 Net
36644.1814 Tax
14620.5723 Interest
        26 Iteration Count

6 rows selected.

SQL> @of

      YEAR      SALES
---------- ----------
      1999
      2000    2474.78
      2001    4333.69

SQL> 
```

In general, you can use any `ORDER BY` specification as long as it produces a unique order among cells that qualify the left side cell reference. Expressions in the `ORDER BY` of a rule can involve constants, measures and dimension keys, and you can specify the ordering options `[ASC | DESC] [NULLS FIRST | NULLS LAST]` to get the order you want.