Performing Location-Based Analysis

Purpose

This module shows you how to use Oracle Locator or Oracle Spatial, and Oracle Workspace Manager for location-based analysis on current and proposed data.

Topics

This module will discuss the following topics:

Place the cursor on this icon to display all screenshots. You can also place the cursor on each icon to see only the screenshot associated with it.

Overview

Back to List

Oracle Locator and Oracle Workspace Manager are features of Oracle10i Database, Standard and Enterprise Editions. Oracle Locator provides an integrated set of functions and procedures to efficiently store, manage, query and analyze spatial data in an Oracle database, using standard SQL. Oracle Workspace Manager allows current, proposed and historical values of the data to be managed in the same database.

Oracle Spatial, an option for Oracle10i Database, Enterprise Edition, augments Oracle Locator with additional high-end spatial functionality including: functions such as buffer generation, spatial aggregates, area calculations, and more; linear referencing; coordinate systems transformations; topology data model; and support for geo-referenced raster data.

Scenario

MyCompany has several major warehouses. It needs to locate its customers who are near a given warehouse, to inform them of new advertising promotions. To locate its customers and perform location-based analysis, MyCompany must store location data for both its customers and warehouses.

This module uses the **`customers`** and **`warehouses`** tables in the OE schema.

The **`customers`** table has the following fields:

Column	Data Type
customer_id	NUMBER(6)
cust_first_name	VARCHAR2(20)
cust_last_name	VARCHAR2(20)
cust_address	cust_address_typ
phone_numbers	phone_list_typ
nls_language	VARCHAR2(3)
nls_territory	VARCHAR2(30)
credit_limit	NUMBER(9,2)
cust_email	VARCHAR2(30)
account_mgr_id	NUMBER(6)
cust_geo_location	MDSYS.SDO_GEOMETRY

The **warehouses** table has the following fields:

Column	Data Type
warehouse_id	NUMBER(3)
warehouse_spec	SYS.XMLTYPE
warehouse_name	VARCHAR2(35)
location_id	NUMBER(4)
wh_geo_location	MDSYS.SDO_GEOMETRY

Oracle's Data Types

Oracle's data types include:

- Numbers (NUMBER)
- Characters (VARCHAR2)

- Dates (DATE)
- Spatial data (MDSYS.SDO_GEOMETRY). A location can be stored as a point in an SDO_GEOMETRY column of a table. The customer's location is associated with longitude and latitude values on the Earth's surface—for example, -63.13631, 52.485426.

Prerequisites

Back to List

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

☐ **1.** Install Oracle Database 10g.

☐ **2.** Download the spatial.zip into your working directory.

Load New Customers and Their Locations

Back to Topic List

In this module it is assumed that you have already loaded the Order Entry (OE) schema, which contains the **`customers`** and **`warehouses`** tables. Perform the following tasks:

1. Load the Location Data
2. Add New Customers and Their Locations to the **`customers`** Table
3. Add Metadata to the **`user_sdo_geom_metadata`** View

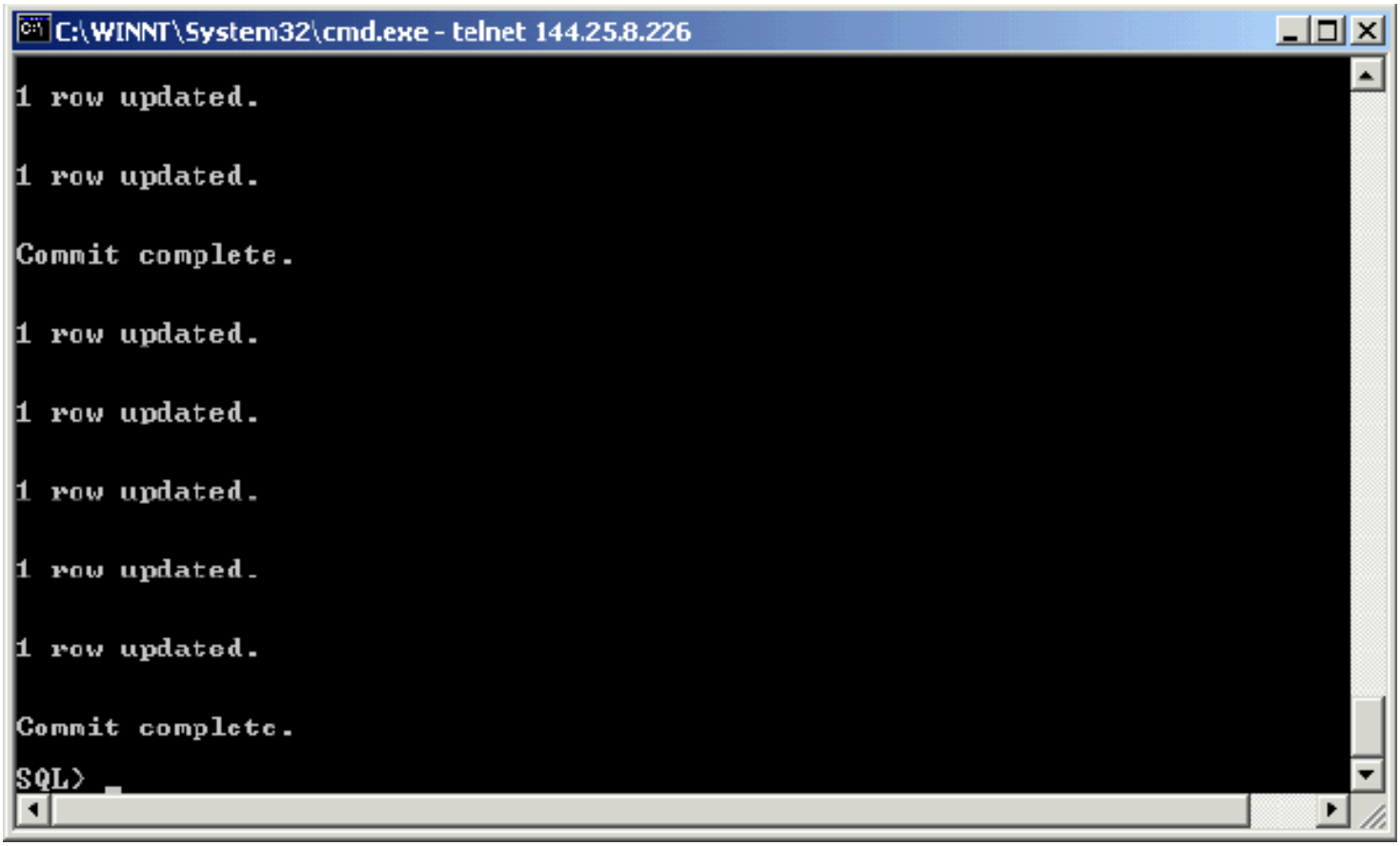
1. Loading the Location Data

Back to List

Several customers and warehouses in the Order Entry schema have location values of NULL. To supply locations for those customers and warehouses, perform the following steps:

☐ **1.** Open a **SQL*Plus** session from your working directory **/home/oracle/wkdir** and execute the following commands:

```
connect oe/oe@<sid>
@loc_updates
```



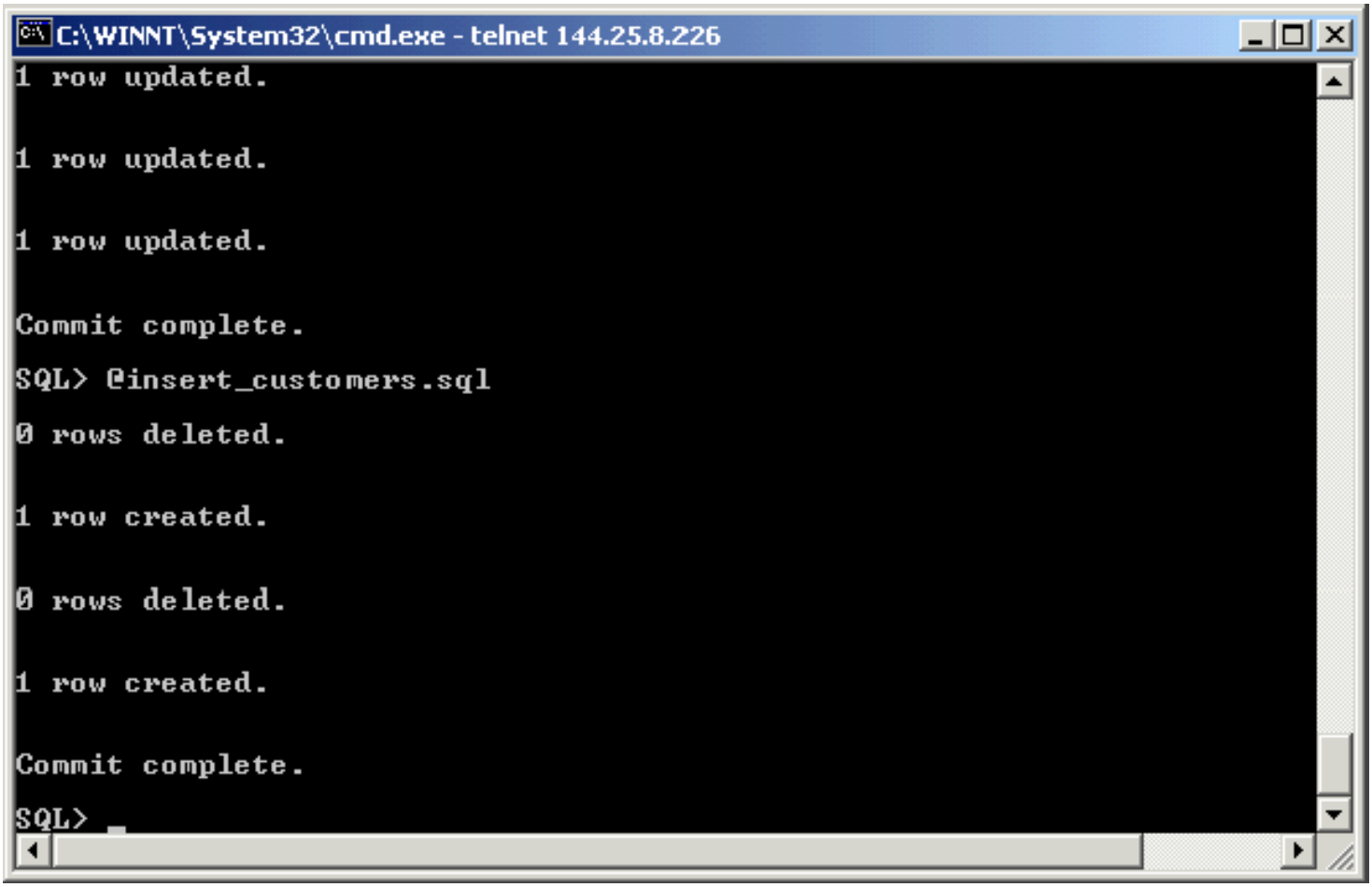
2. Adding New Customers and Their Locations to the `customers` Table

Back to List

A transactional insert is used to add new customers and their locations to the **`customers`** table. A customer's location can be stored as a point in an `SDO_GEOMETRY` column in a table. The customer's location is associated with longitude and latitude values on the Earth's surface (for example, -63.136, 52.4854). Oracle Locator and Oracle Spatial require that you place the longitude value before the latitude value. In the `INSERT` statement below, an `SDO_GEOMETRY` constructor is used to insert the point location.

To add a new customer and his or her location to the **`customers`** table, perform the following steps:

☐ **1.** From a **SQL*Plus** session logged on to the OE schema, run **@insert_customers.sql** .



Description of SDO_GEOMETRY Constructor

Below is a brief description of the SDO_GEOMETRY constructor that is populated in this exercise:

```
MDSYS.SDO_GEOMETRY ( 2001 , 8307 ,
MDSYS.SDO_POINT_TYPE (-63.13631,52.485424,NULL), NULL,NULL )
```

The elements of the syntax have the following meanings:

2001	This is the SDO_GTYPE attribute and it is set to 2001 when storing a two-dimensional single point such as a customer's location.

8307	This is the spatial reference system ID (SRID): a foreign key to an Oracle dictionary table (MDSYS.CS_SRS) that contains all the supported coordinate systems. It is important to associate your customer's location to a coordinate system. In this example, 8307 corresponds to "Longitude / Latitude (WGS 84)."
MDSYS.SDO_POINT_TYPE	This is where you store your longitude and latitude values within the SDO_GEOMETRY constructor. Note that you can store a third value also, but for these modules, all the customer data is two-dimensional.
NULL, NULL	The last two null values are beyond the scope of this module. You can construct very powerful location-based queries without understanding the last two fields of the SDO_GEOMETRY constructor. For more information on all the fields of the SDO_GEOMETRY object, please refer to the *Oracle Spatial User's Guide and Reference* . For now, these last two fields should be set to NULL.

3. Add Metadata to the `user_sdo_geom_metadata` View

Back to List

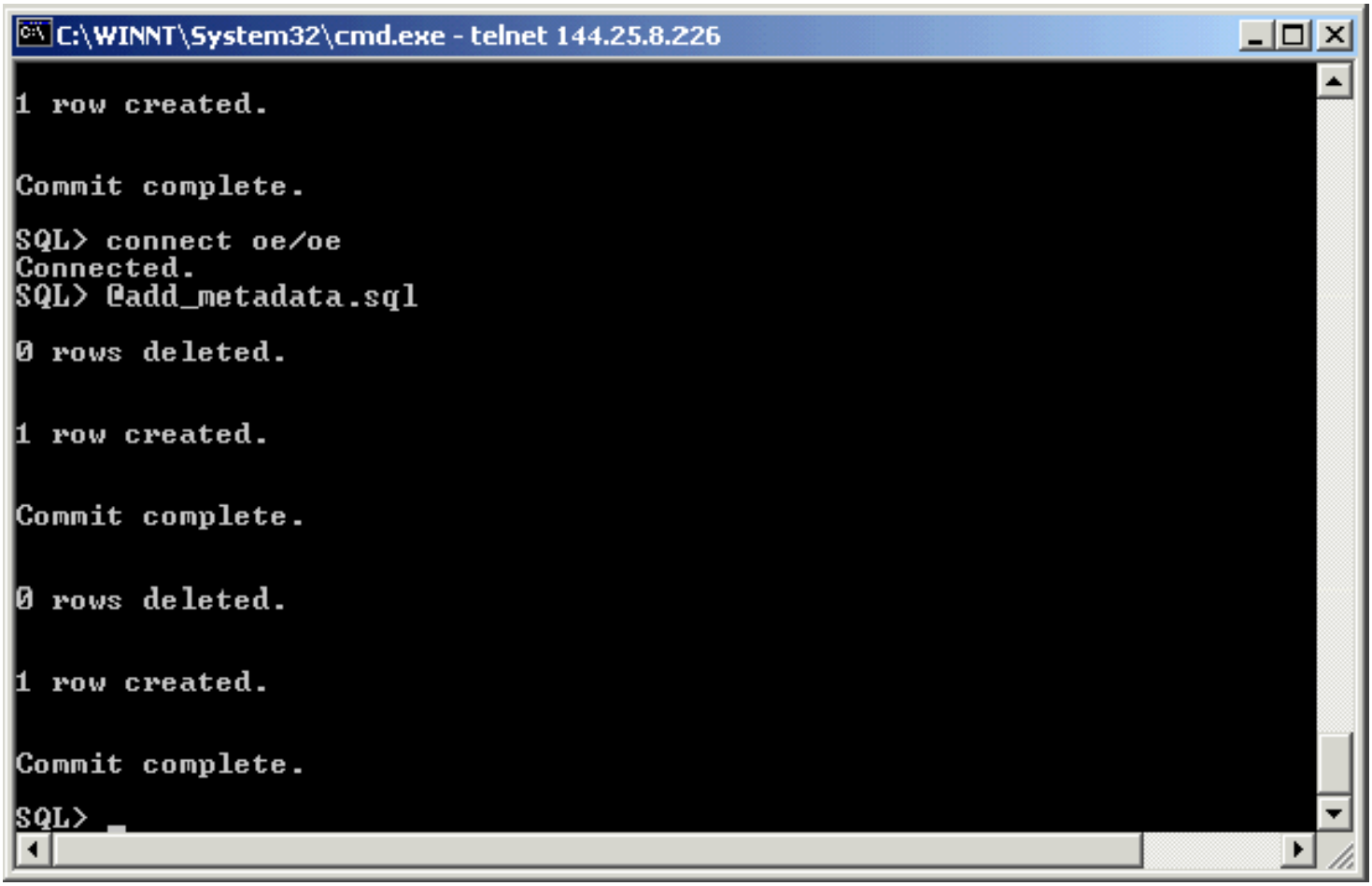
Before creating a spatial index, you must add metadata for the **`customers`** and **`warehouses`** tables to the **`USER_SDO_GEOM_METADATA`** view.

Note:

- Add one row for every SDO_GEOMETRY column
- The SDO_GEOMETRY column for **`customers`** is `cust_geo_location` .
- The SDO_GEOMETRY column for **`warehouses`** is `wh_geo_location` .

To add metadata for customers and warehouses, perform the following steps:

☐ **1.** From a **SQL*Plus** session logged on to the OE schema, run **@add_metadata.sql** .



Here is a description of the information that was inserted:

TABLE_NAME	This is the name of the table which contains the spatial data.
COLUMN_NAME	This is the name of the SDO_GEOMETRY column which stores the spatial data
MDSYS.SDO_DIM_ARRAY	This is a constructor which holds the MDSYS.SDO_DIM_ELEMENT object, which in turn stores the extents of the spatial data in each dimension (-180.0, 180.0), and a tolerance value (0.005). The tolerance is a round-off error value used by Oracle Spatial, and is in meters for longitude and latitude data. In this example, the tolerance is 5 mm.

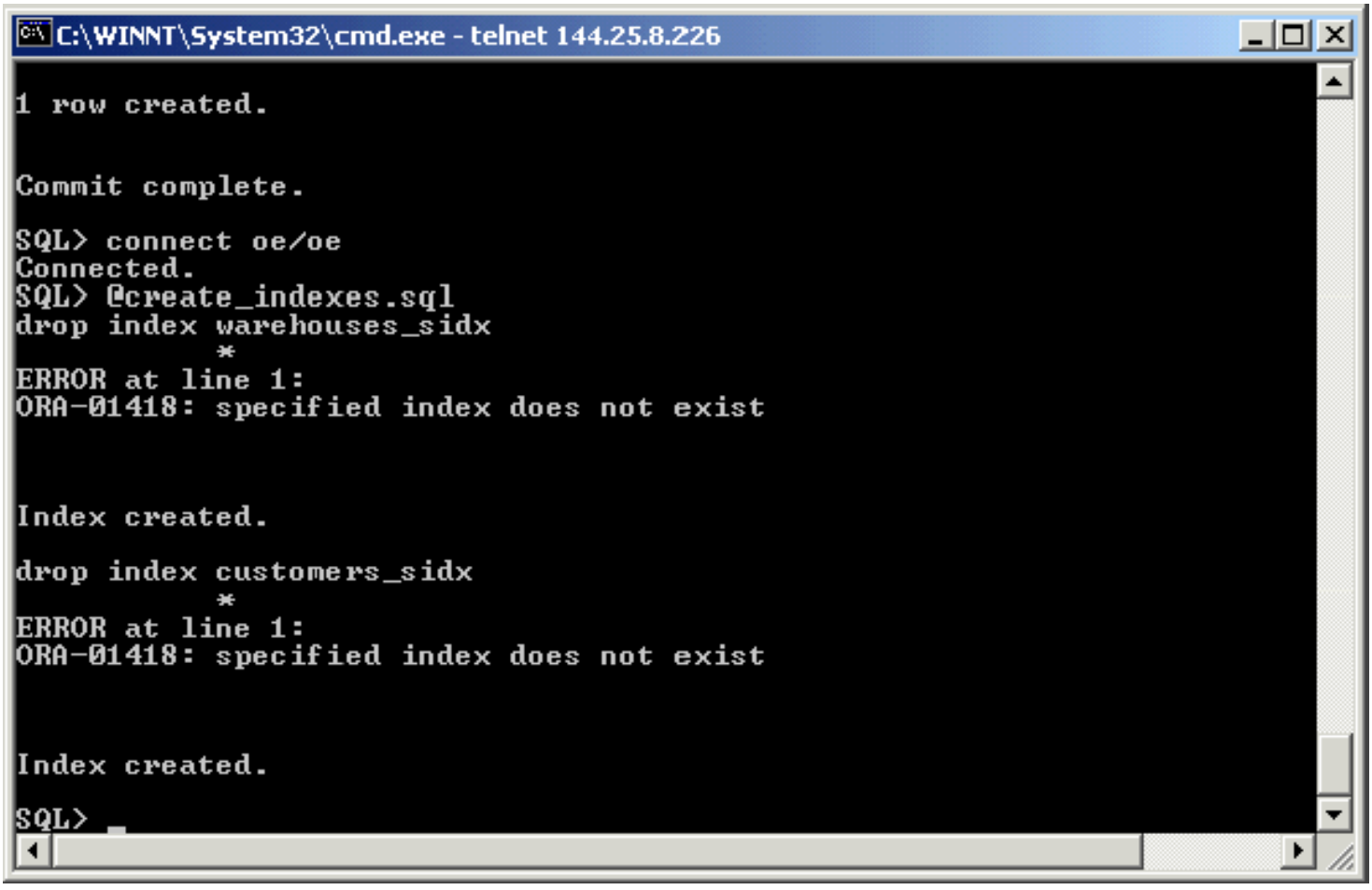
8307	This is the spatial reference system id (SRID): a foreign key to an Oracle dictionary table (`MDSYS.CS_SRS`) that contains all the supported coordinate systems. It is important to associate your customer's location to a coordinate system. In this example, 8307 corresponds to "Longitude / Latitude (WGS 84)."

Creating a Spatial Index on a Geometry Column

Back to Topic List

You are now ready to create spatial indexes for **`customers`** and **`warehouses`** :

☐ **1.** From a SQL*Plus session logged on to the OE schema, run **`@create_indexes.sql`** to create the indexes.



LAYER_GTYPE	This parameter works both as a constraint and as a hint to the optimizer. If the parameter LAYER_GTYPE =POINT is used, checks are made to ensure that all geometries are points, as well as the parameter ensures optimized processing of point data. **`customers`** and **`warehouses`** both contain point geometries only.

Performing Location-Based Queries

Back to Topic List

In that pages that follow, you will learn how to perform the following types of location-based queries:

1. Using the Spatial Index to Find the Five Nearest Neighbors to a Warehouse, no Additional Constraints
2. Using the Spatial Index to Find the Five Nearest Neighbors in a Location, with Additional Constraints
3. Using the Spatial Index to Identify the Set of Locations that are within some Specified Distance of another Location

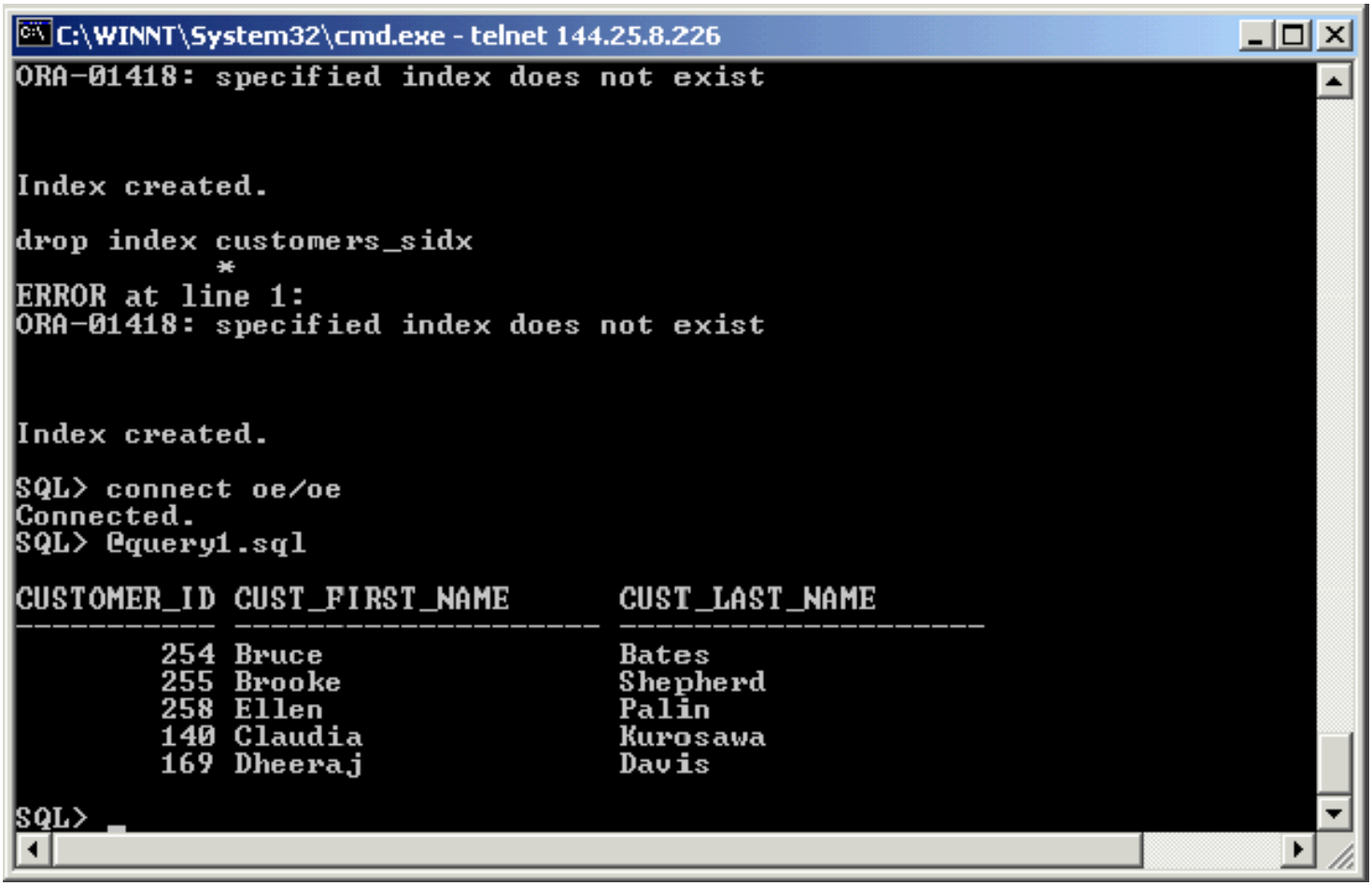
1. Using the Spatial Index to Find the Five Nearest Neighbors to a Warehouse, no Additional Constraints

Back to List

Query 1: Find the five customers closest to the warehouse whose warehouse ID is 2.

Perform the following steps:

☐ **1.** From a **SQL*Plus** session logged on to the OE schema, run **`@query1.sql`** :



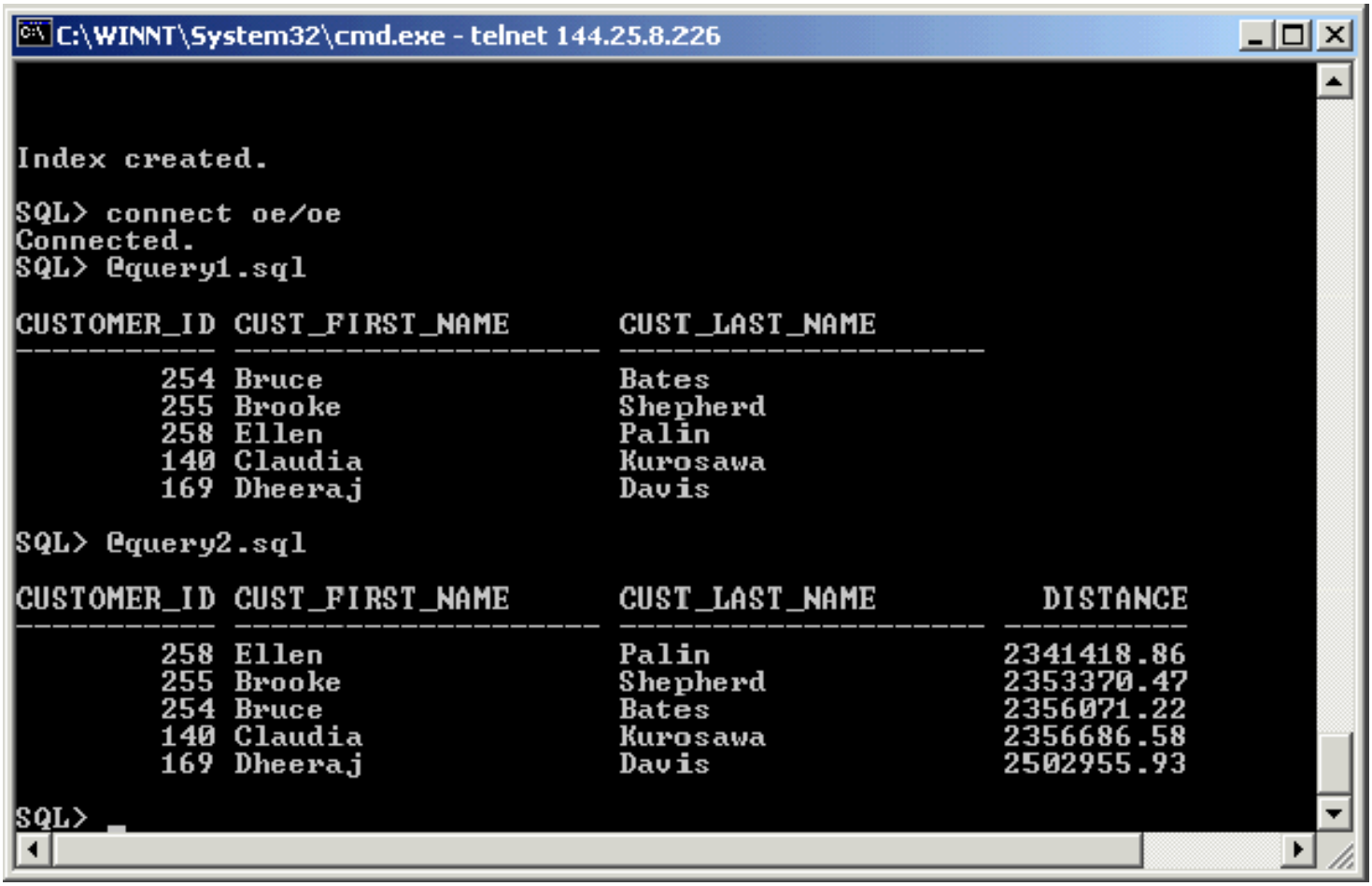
Here is a description of the information that is selected:

- The `/*+ordered*/` hint is a hint to the optimizer, which ensures that the **`warehouses`** table is searched first.
- The `SDO_NN` operator returns the `SDO_NUM_RES` value of the customers from the **`customers`** table who are closest to warehouse 2. The first argument to `SDO_NN` (`c.cust_geo_location` in the example above) is the column to search. The second argument to `SDO_NN` (`w.wh_geo_location` in the example above) is the location you want to find the neighbors nearest to. No assumptions should be made about the order of the returned results. For example, the first row returned is not guaranteed to be the customer closest to warehouse 2. If two or more customers are an equal distance from the warehouse, then either of the customers may be returned on subsequent calls to `SDO_NN` .
- When using the `SDO_NUM_RES` parameter, no other constraints are used in the `WHERE` clause. `SDO_NUM_RES` takes only proximity into account. For example, if you added a criterion to the `WHERE` clause because you wanted the five closest customers that resided in NY, and four of the five closest customers resided in NJ, the query above would return one row. This behavior is specific to the `SDO_NUM_RES` parameter, and its results may not be what you are looking for. You will learn how to find the five closest customers who reside in NY in the discussion of query 3.

Query 2: Find the five customers closest to warehouse 2 and put the results in order of distance

To return the actual distances for the five closest customers, you can use the `SDO_NN_DISTANCE` operator. Perform the following steps:

☐ **1.** From a **SQL*Plus** session logged on to the OE schema, run **@query2.sql** :



Here is a description of the information that is selected:.

- The `SDO_NN_DISTANCE` operator is an ancillary operator to the `SDO_NN` operator; it can only be used within the `SDO_NN` operator. The argument for this operator is a number that matches the number specified as the last argument of `SDO_NN` ; in this example it is 1. There is no hidden meaning to this argument, it is simply a tag. If `SDO_NN_DISTANCE( )` is specified, you can order the results by distance and guarantee that the first row returned is the closest. If the data you are querying is stored as longitude and latitude, the default unit for `SDO_NN_DISTANCE` is meters.
- The `SDO_NN` operator also has a `UNIT` parameter that determines the unit of measure returned by `SDO_NN_DISTANCE` . However, it is not used in this example.
- The `ORDER BY DISTANCE` clause ensures that the distances are returned in order, with the shortest distance first.

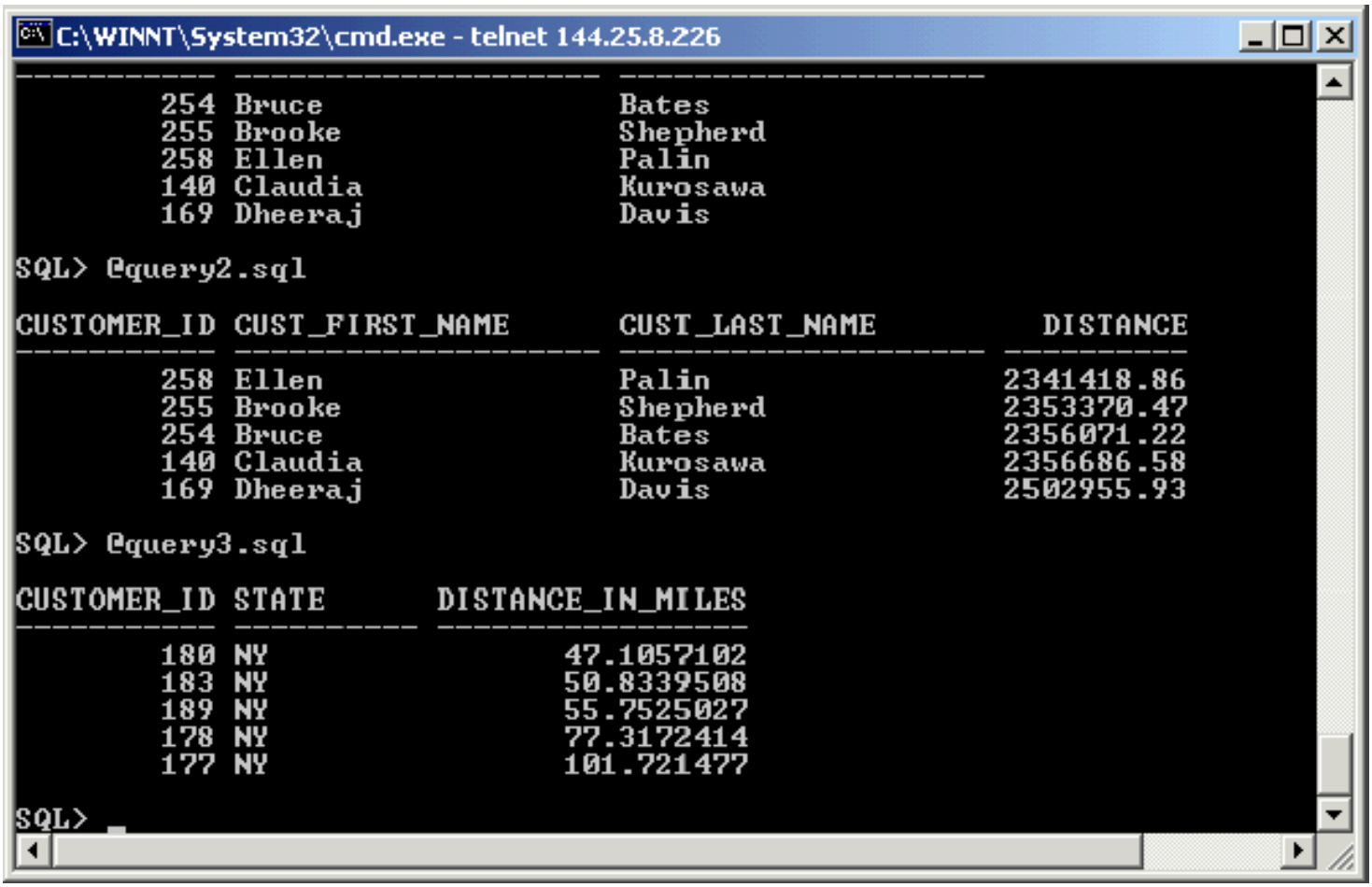
2. Using the Spatial Index to Find the Five Nearest Neighbors in a Location, with Additional Constraints

Back to List

Query 3: Find the five customers closest to warehouse 3 who reside in NY state, put the results in order of distance, and give the distance in miles

Perform the following steps:

☐ **1.** From a **SQL*Plus** session logged on to the OE schema, run **@query3.sql :**



Here is a description of the information that was selected:

- `SDO_BATCH_SIZE` is a tunable parameter that may affect your query's performance. `SDO_NN` internally calculates that number of distances at a time. The initial batch of rows returned may not satisfy the constraints in the `WHERE` clause, so the number of rows specified by `SDO_BATCH_SIZE` is continuously returned until all the constraints in the `WHERE` clause are satisfied. You should choose a `SDO_BATCH_SIZE` that initially returns the number of rows likely to satisfy the constraints in your `WHERE` clause.
- The `UNIT` parameter used within the `SDO_NN` operator specifies the unit of measure of the `SDO_NN_DISTANCE` parameter. The default unit is the unit of measure associated with the data. For longitude and latitude data, the default is meters.
- `c.cust_address.state_province = 'NY'` and `rownum < 6` are the additional constraints in the `WHERE` clause. The `rownum < 6` clause is necessary to limit the number of results returned to fewer than 6.
- The `ORDER BY DISTANCE_IN_MILES` clause ensures that the distances are returned in order, with the shortest distance first and the distances measured in miles.

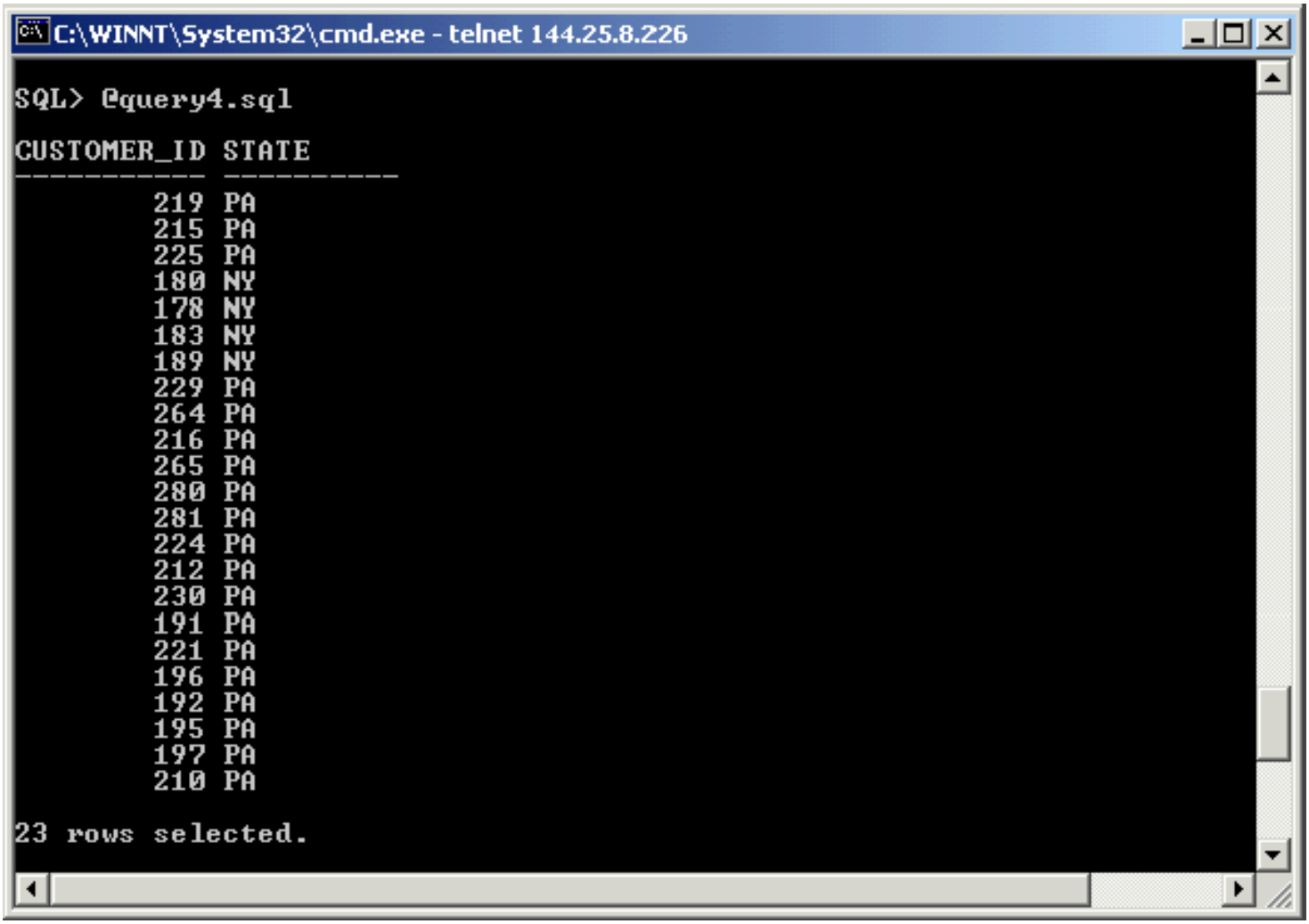
3. Using the Spatial Index to Identify the Set of Locations that are within some Specified Distance of another Location

Back to List

Query 4: Find all the customers within 100 miles of warehouse 3

Perform the following steps:

☐ **1.** From a **SQL*Plus** session logged on to the OE schema, run **@query4.sql** :



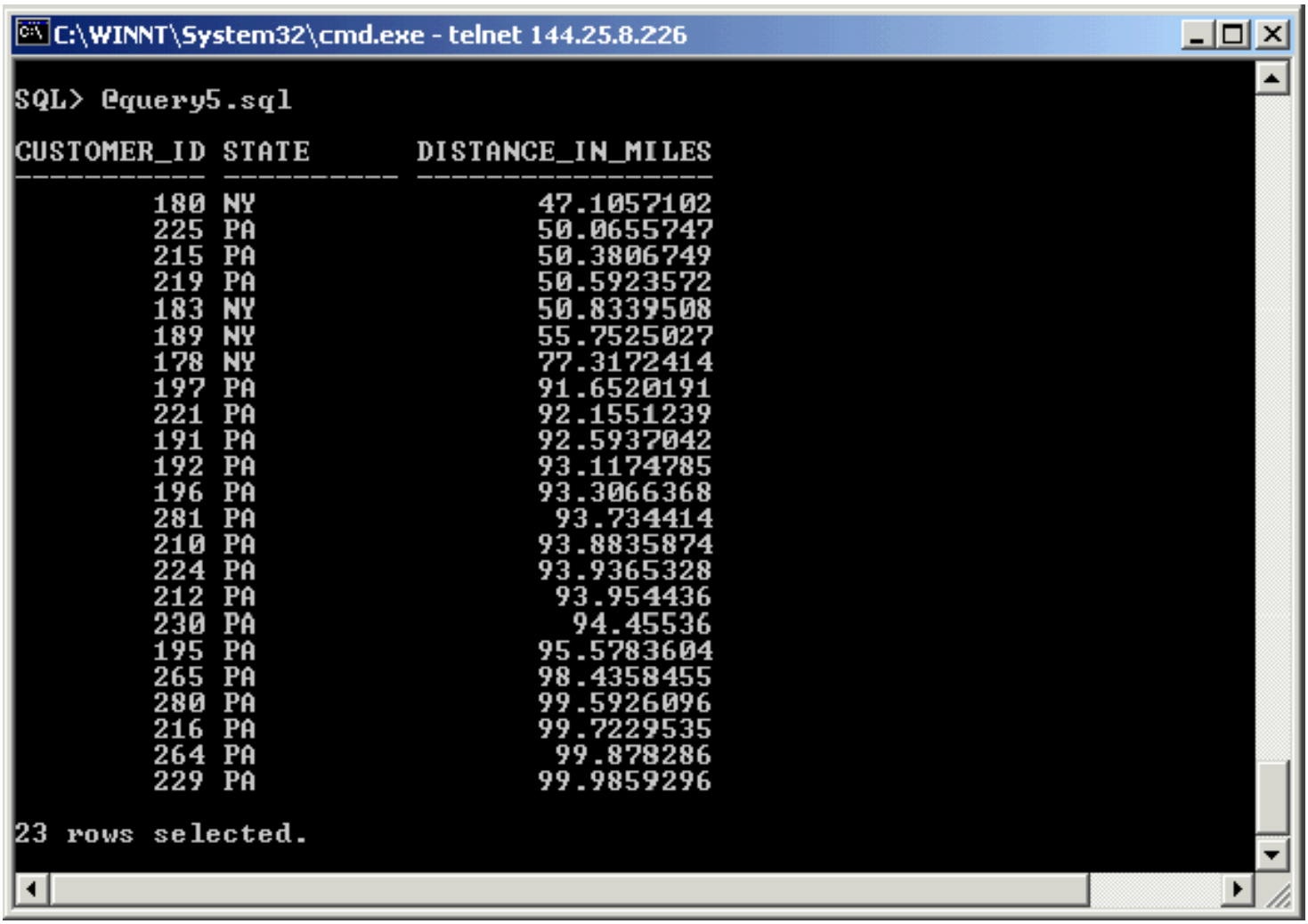
Here is a description of the information that was selected:

- The SDO_WITHIN_DISTANCE operator returns the customers from the **customers** table that are within 100 miles of warehouse 3. The first argument to SDO_WITHIN_DISTANCE (c.cust_geo_location in the example above) is the column to search. The second argument to SDO_WITHIN_DISTANCE (w.wh_geo_location in the example above) is the location you want to determine the distances from. No assumptions should be made about the order of the returned results. For example, the first row returned is not guaranteed to be the customer closest to warehouse 3.
- The DISTANCE parameter used within the SDO_WITHIN_DISTANCE operator specifies the distance value; in this example it is 100.
- The UNIT parameter used within the SDO_WITHIN_DISTANCE operator specifies the unit of measure of the DISTANCE parameter. The default unit is the unit of measure associated with the data. For longitude and latitude data, the default is meters; in this example, it is miles.

Query 5: Find all the customers within 100 miles of warehouse 3, put the results in order of distance, and give the distance in miles

Perform the following steps:

☐ **1.** From a **SQL*Plus** session logged on to the OE schema, run **@query5.sql** :



Here is a description of the information that was selected:

- The SDO_GEOM.SDO_DISTANCE function computes the exact distance between the customer's location and warehouse 3. The first argument to SDO_GEOM.SDO_DISTANCE (c.cust_geo_location in the example above) contains the customer's location whose distance from warehouse 3 is to be computed. The second argument to SDO_WITHIN_DISTANCE (w.wh_geo_location in the example above) is the location of warehouse 3, whose distance from the customer's location is to be computed.
- The third argument to SDO_GEOM.SDO_DISTANCE (0.005) is the tolerance value. The tolerance is a round-off error value used by Oracle Spatial. The tolerance is in meters for longitude and latitude data. In this example, the tolerance is 5 mm.
- The UNIT parameter used within the SDO_GEOM.SDO_DISTANCE parameter specifies the unit of measure of the distance computed by the SDO_GEOM.SDO_DISTANCE function. The default unit is the unit of measure associated with the data. For longitude and latitude data, the default is meters. In this example it is miles.
- The ORDER BY DISTANCE_IN_MILES clause ensures that the distances are returned in order, with the shortest distance first and the distances measured in miles.

Creating and Using a Function-Based Index

Back to Topic List

A function-based index allows indexes to be built on the results of a function that returns a SDO_GEOMETRY object. It is a powerful mechanism which enables location-based functionality without requiring a SDO_GEOMETRY column in a table. The function-based index is intended for use on tables with columns that store longitude and latitude data.

How to Create and Use a Function-Based Index

The steps to create and use a function-based index are as follows:

Setup: Updating the **warehouses** Table

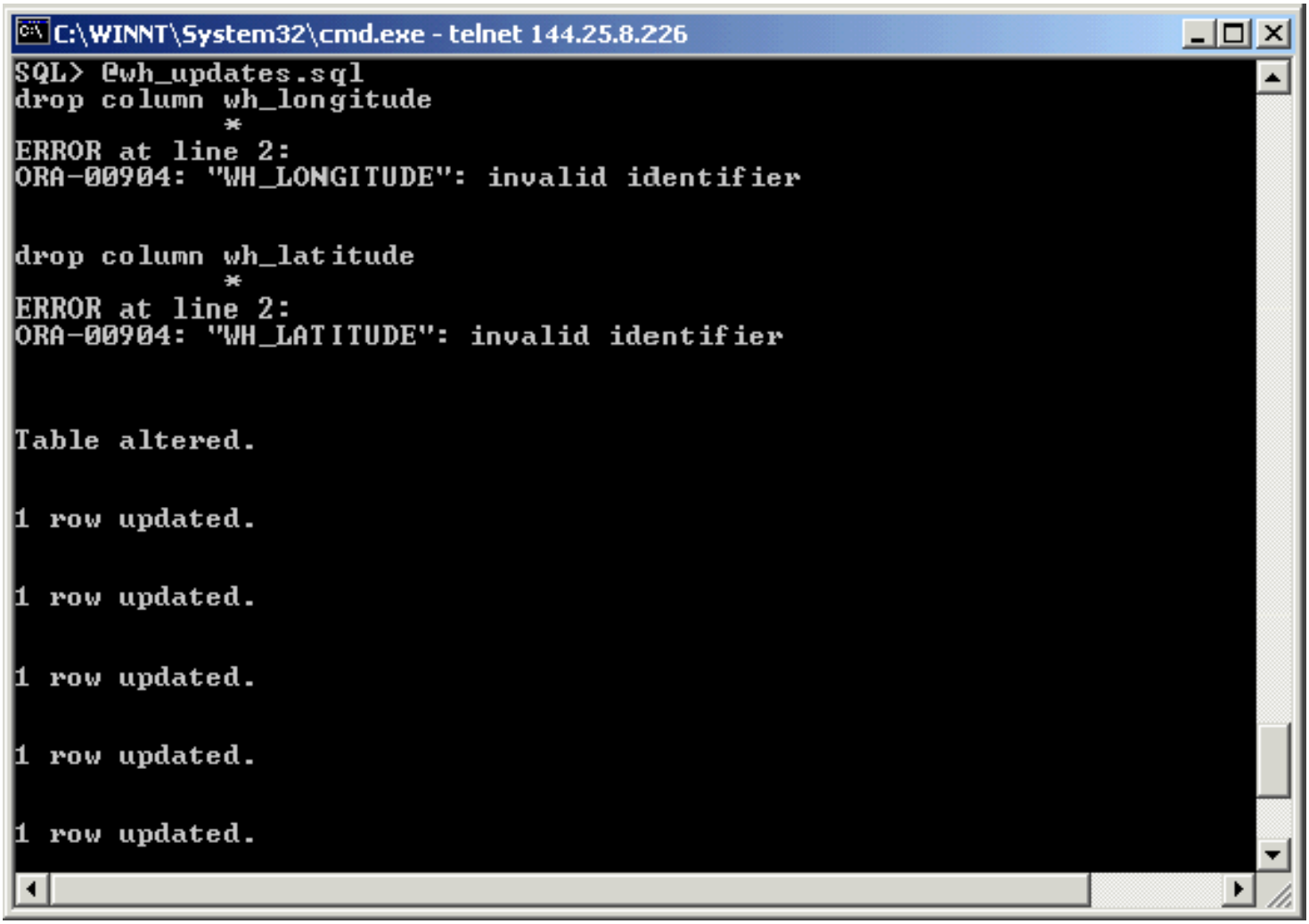
1. Creating a Function that Returns a SDO_GEOMETRY Object
2. Adding Metadata to the **user_sdo_geom_metadata** View for a Function
3. Creating a Function-Based Spatial Index
4. Using the Function-Based Spatial Index

Setup: Updating the warehouses Table

Back to List

Before you continue, you must run the wh_updates.sql script. This script adds the wh_longitude and wh_latitude columns to the **warehouses** table. wh_longitude and wh_latitude are columns of type NUMBER that will be spatially indexed using the function-based index.

☐ **1.** From a **SQL*Plus** session logged on to the OE schema, run **@wh_updates.sql** to load the data.



1. Creating a Function that Returns a SDO_GEOMETRY Object

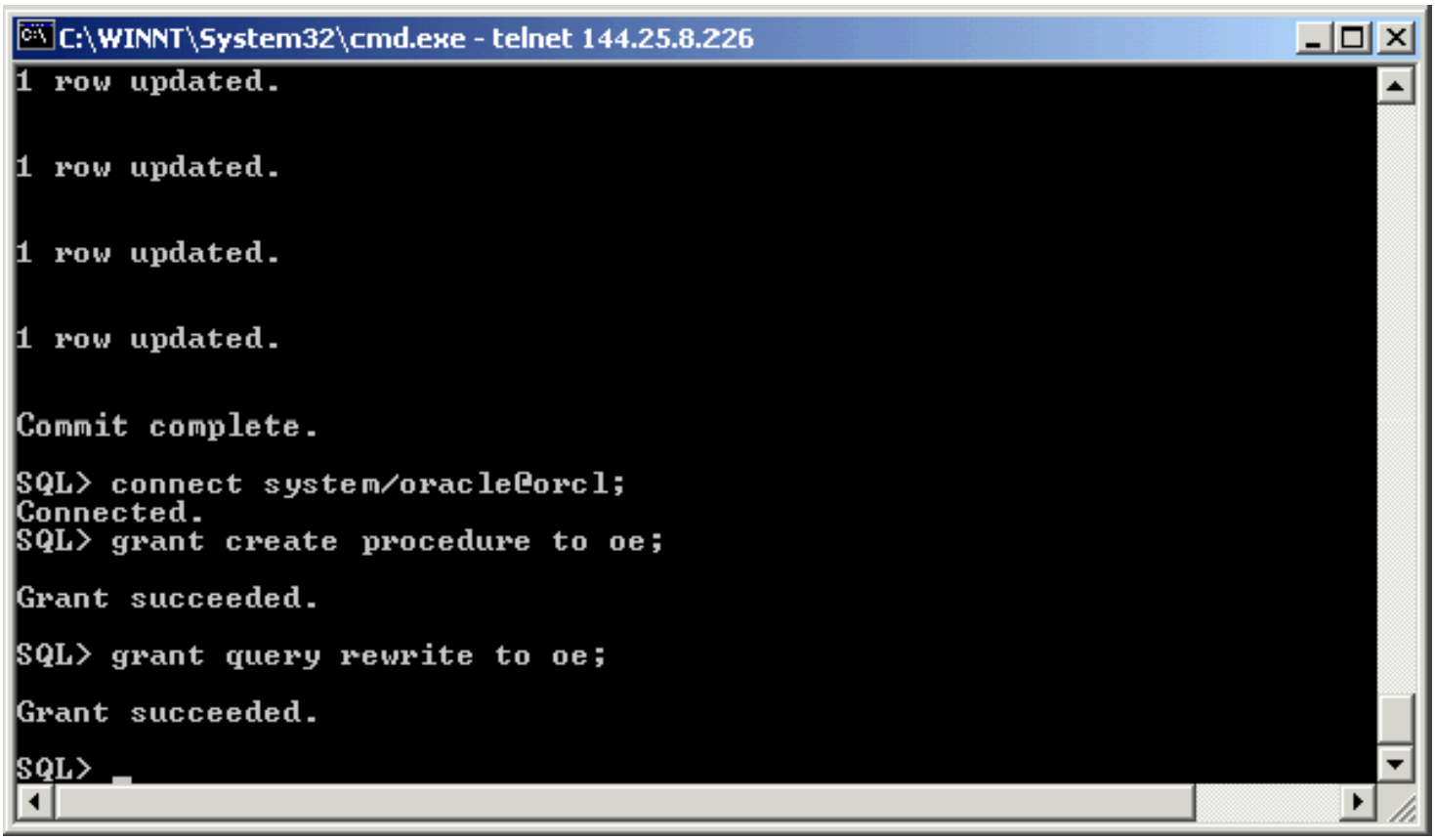
Back to List

To create a function that returns a SDO_GEOMETRY object, perform the following steps:

☐ **1.** Before you can create the function, you need to ensure that OE has the privilege to do so. At the same time, grant OE the privilege, which allows the Oracle optimizer to use the function-based index. Specify the following commands:

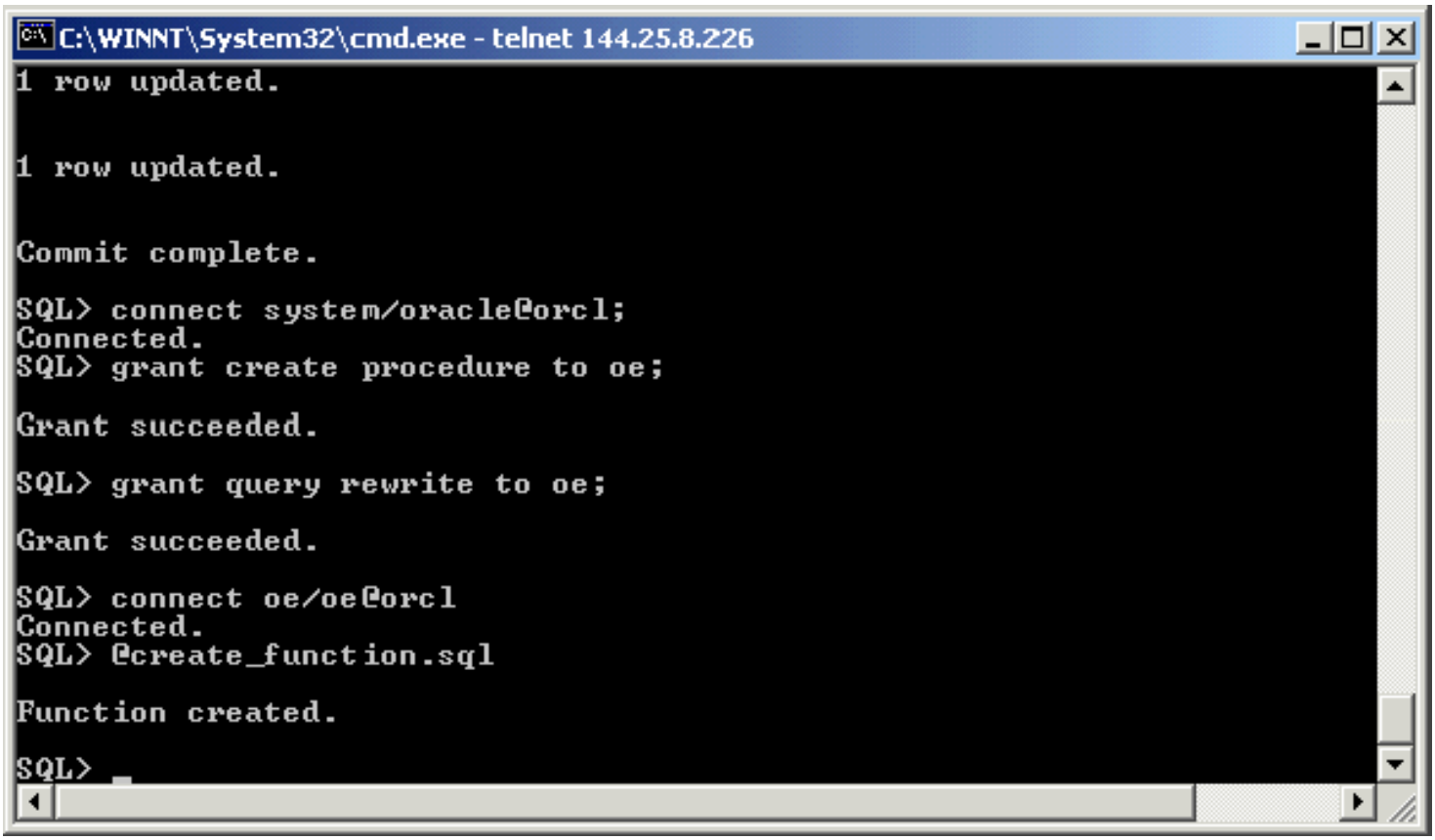
```
connect system/<password>@<sid>;
grant create procedure to oe;
```

```
Grant query rewrite to oe;
```



☐ **2.** Connect again to your instance as **oe / oe** .

☐ **3.** From a **SQL*Plus** session logged on to the OE schema, run **@create_function.sql** :



Description of the Executed SQL Query

The function must be declared as `DETERMINISTIC` , otherwise the optimizer may not use the most optimal plan. This is true for any function that returns an object. The `GET_GEOM` function takes two arguments of type `NUMBER` and returns a `MDSYS.SDO_GEOMETRY` object.

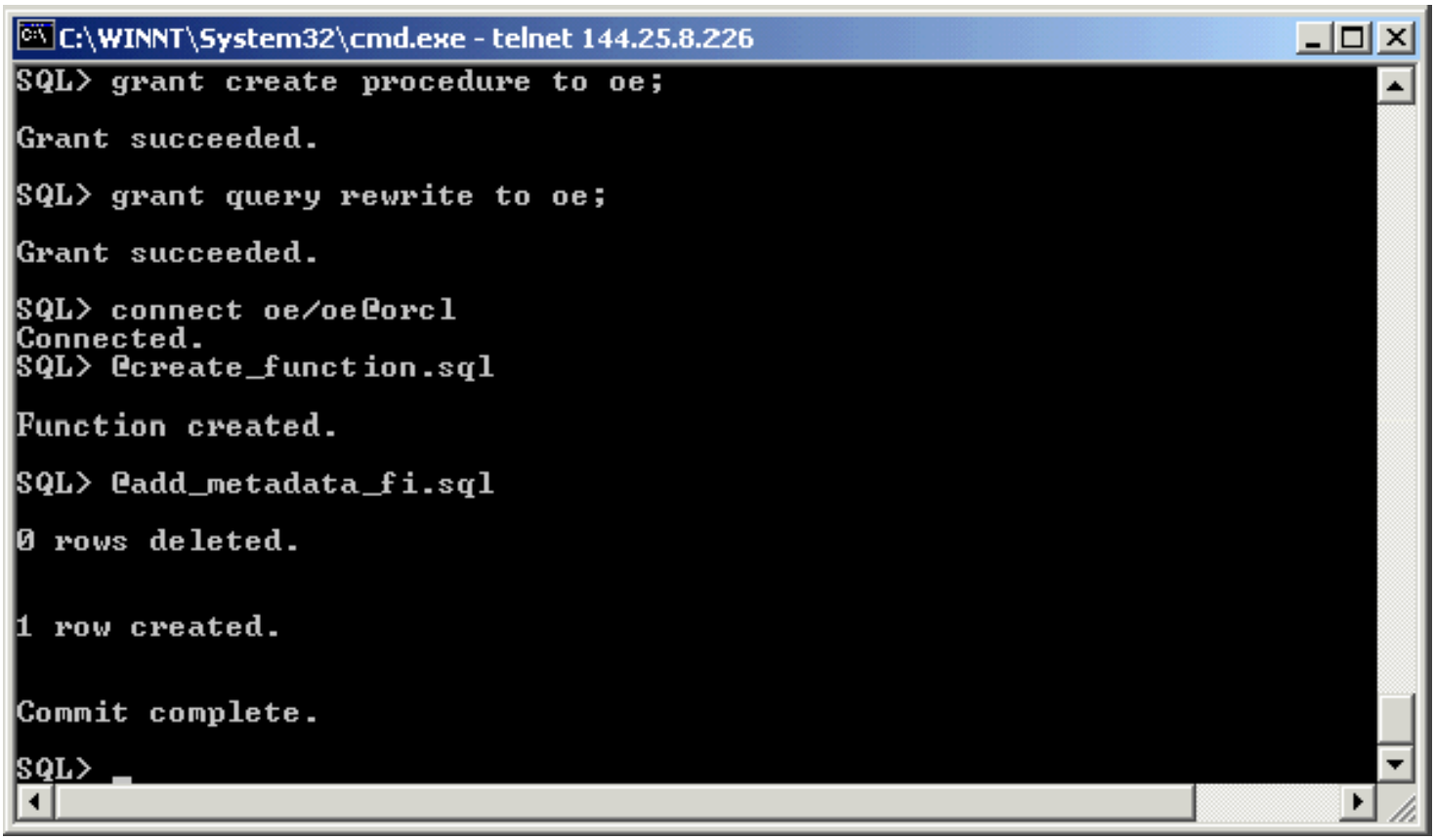
2. Adding Metadata to the `user_sdo_geom_metadata` View for a Function

Back to List

Before creating a spatial index, you must add metadata to the `user_sdo_geom_metadata` view. One row is added for a function that returns the `SDO_GEOMETRY` object.

To add metadata for a function that returns a `SDO_GEOMETRY` object, perform the following steps:

☐ **1.** From a **SQL*Plus** session logged on to the OE schema, run **@add_metadata_fi.sql** :



Description of the Executed SQL Query

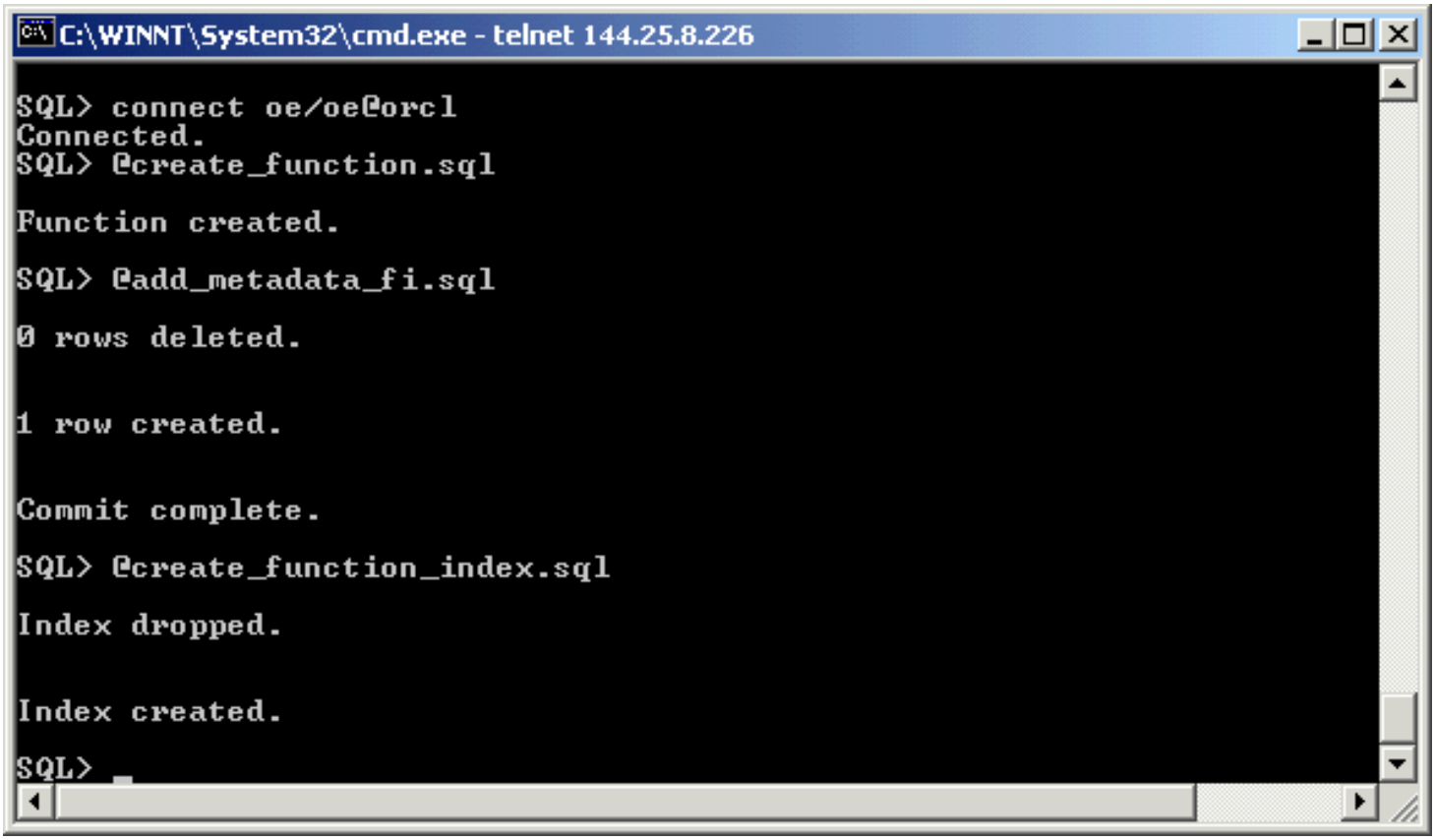
The `GET_GEOM` function is loaded into the `COLUMN_NAME` column of **`user_sdo_geom_metadata`** . The owner of the function (`OE`) is included with the function call. The arguments to the `GET_GEOM` function (`wh_longitude` , `wh_latitude`) are the longitude and latitude columns in the **`warehouses`** table.

3. Creating a Function-Based Spatial Index

Back to List

To create a function-based index, a user must have Query Rewrite privileges. Because you have already granted these privileges to the `OE` user, you can go ahead and create the index. To create a function-based index, perform the following steps:

☐ **1.** From a **SQL*Plus** session logged on to the OE schema, run **@create_function_index.sql** :



Description of the Executed SQL Query

The GET_GEOM function is called as the COLUMN_NAME argument of the CREATE_INDEX statement. The arguments to the GET_GEOM function (wh_longitude , wh_latitude) are the longitude and latitude columns in the **warehouses** table.

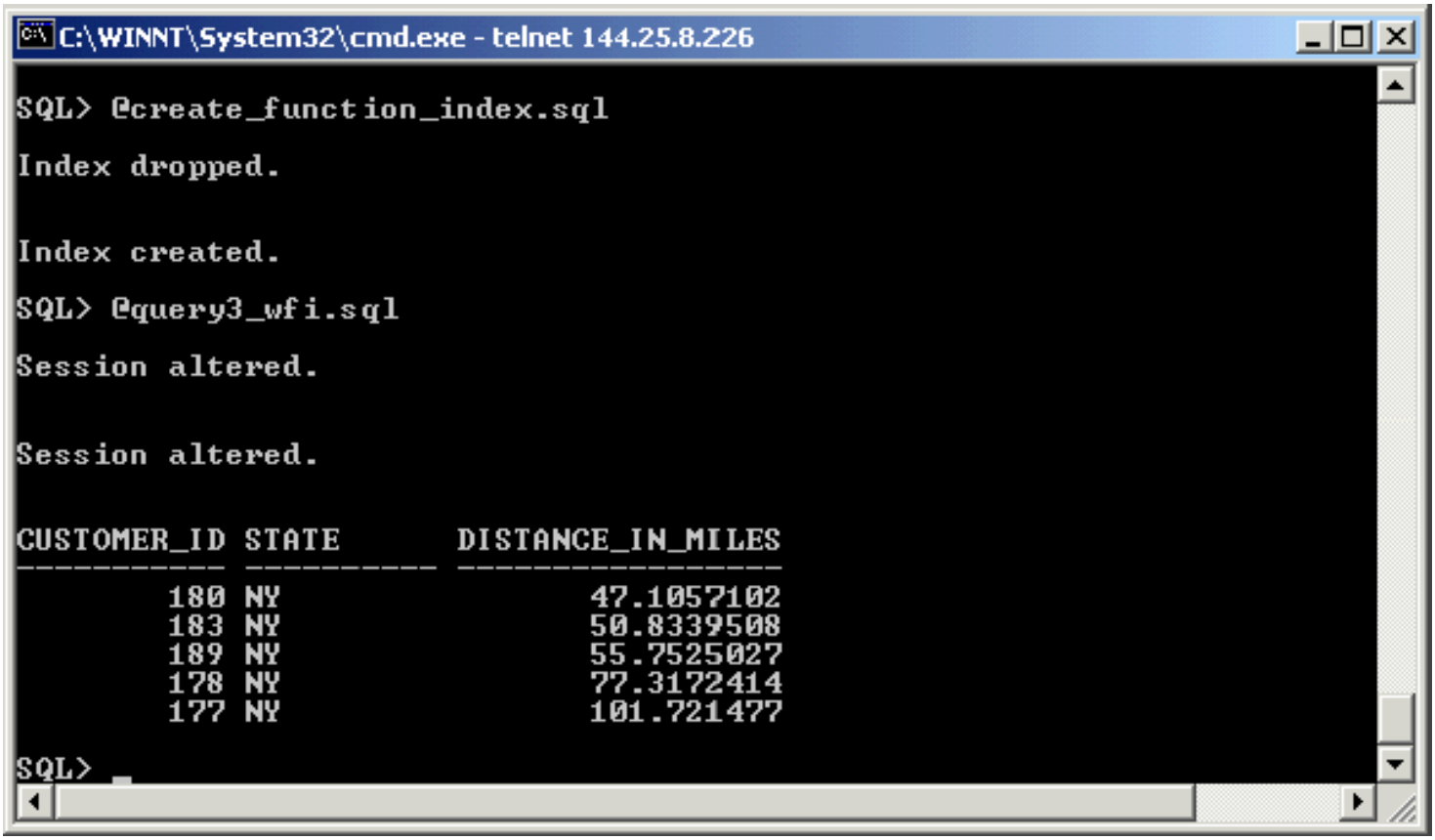
4. Using the Function-Based Spatial Index

Back to List

Perform the following steps:

☐ **1.** To perform queries with function-based indexes, session privileges are required. QUERY_REWRITE_ENABLED must be set to True and QUERY_REWRITE_INTEGRITY must be set to Trusted for the session. Now rerun query 3. However, this time, use the function-based spatial index on the second argument to the SDO_NN operator instead of an SDO_GEOMETRY column.

☐ **2.** From a **SQL*Plus** session logged on to the OE schema, run **@query3_wfi.sql** . The results are as follows:



Description of the Executed SQL Query

The `GET_GEOM` function is called as the second argument to the `SDO_NN` operator. The arguments to the `GET_GEOM` function (`wh_longitude` , `wh_latitude`) are the longitude and latitude columns in the **`warehouses`** table.

Analyzing Current and Proposed Location Data Using Workspace Manager

Back to Topic List

MyCompany is planning to build another warehouse to provide better service to its customers. Two potential sites are under consideration. MyCompany wants to analyze these prospective sites using existing SQL applications. To do this, data for both prospective sites must be entered into the production warehouse table. However, it is important to isolate the prospective site data so it does not impact the work of employees who are not part of the site selection teams. Isolating prospective site data will also allow the two site selection teams to work concurrently.

Oracle Workspace Manager enables current, proposed and historical row versions to exist in the same table. It requires no changes to application SQL (DML). It improves concurrency by allowing production users to access current data while other users go to a workspace to create and access proposed and historical data values. Changes made from a workspace can merged into current data as a group. Workspace Manager allows users to see a complete picture of the database in the context of the changes made from a workspace. It frees developers from writing custom code and DBAs from copying and synchronizing multiple copies of database tables or adding application specific metadata to track row

versions.

In this section, you will perform the following:

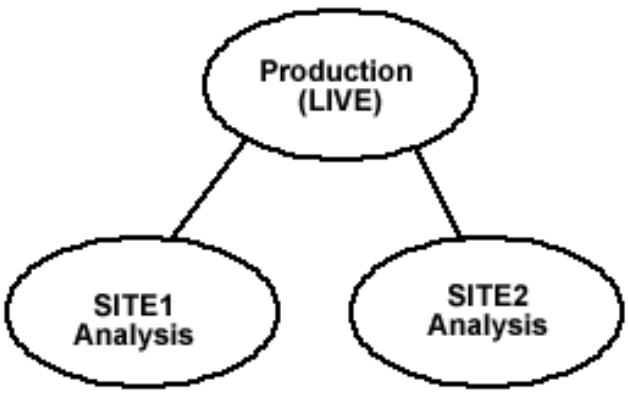
1. Prepare for Warehouse Site Analysis
2. Add Two Proposed Warehouse Locations in a Workspace
3. Determine which Location is Nearest to the Largest Number of Customers
4. Make Chosen Location Data Available to other Users and Clean-up

1. Prepare for Warehouse Site Analysis

Back to List

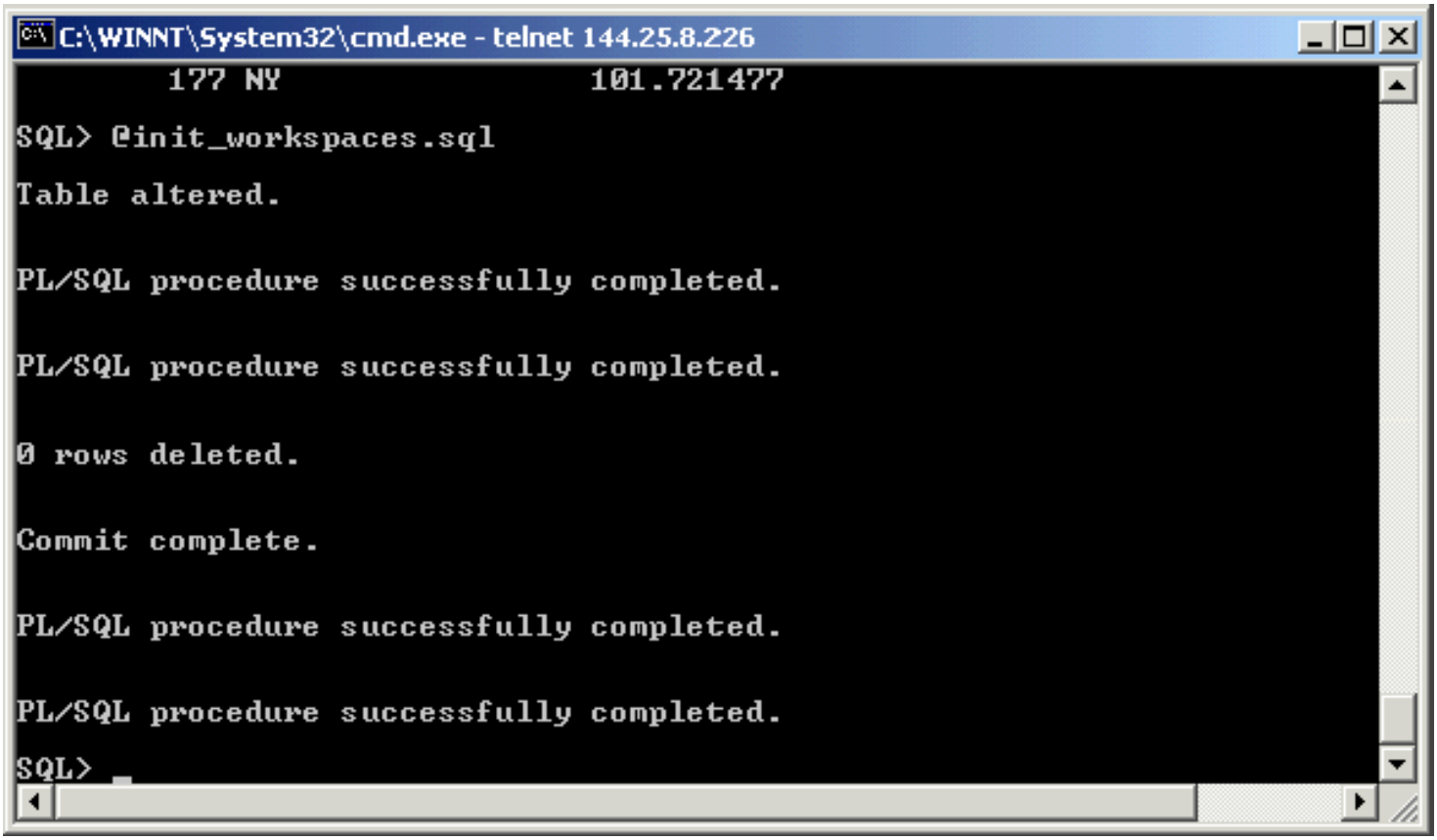
When a table is version-enabled, all rows in the table can support multiple versions of the data. Versioned rows are stored in the same table as the original rows. A workspace is a virtual environment that one or more users can share to make changes to the data in the database. There can be a hierarchy of workspaces in the database. By default, when a workspace is created, it is created from the topmost database workspace, which is always called "LIVE".

You will begin by version-enabling the warehouse table and creating two workspaces, SITE1 and SITE2, off of LIVE, one for each site selection team.



Perform the following steps:

☐ **1.** From a SQL*Plus session logged on to the OE schema, run **@init_workspaces.sql** .



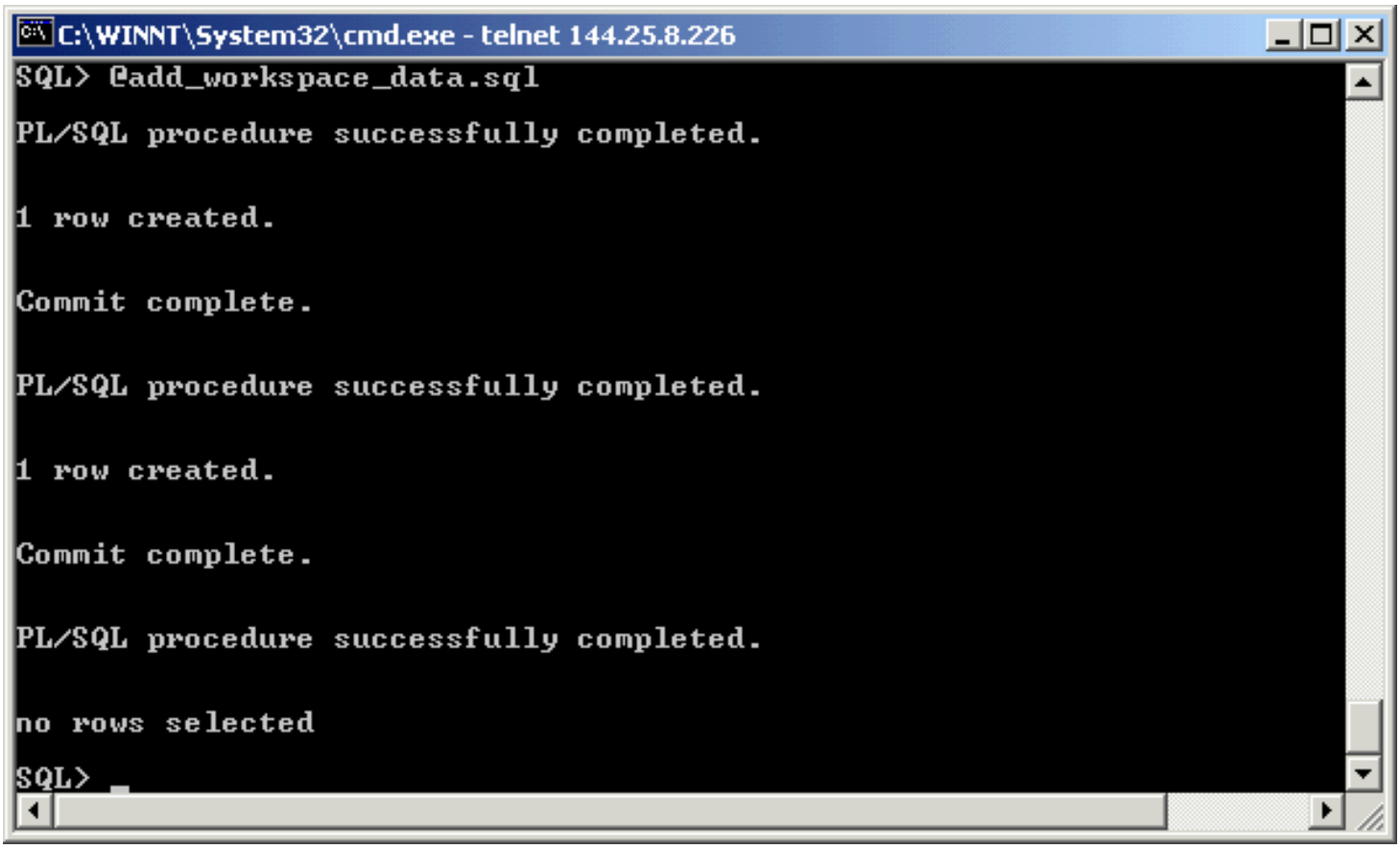
2. Add Two Proposed Warehouse Locations in a Workspace

Back to List

Users enter a workspace to make changes to data. A workspace logically groups collections of row versions from one or more version-enabled tables, and isolates these versions until they are explicitly merged with production data.

Goto workspace SITE1 and add the first proposed warehouse location. Goto workspace SITE2 and add the second proposed warehouse location with same attributes as SITE1 except a different location. Go to workspace LIVE and select all from the warehouse table, the prospective sites do not appear. Users in SITE1 can not see changes made in SITE2 and visa versa. Perform the following steps:

☐ **1.** From a SQL*Plus session logged on to the OE schema, run **@add_workspace_data.sql** .



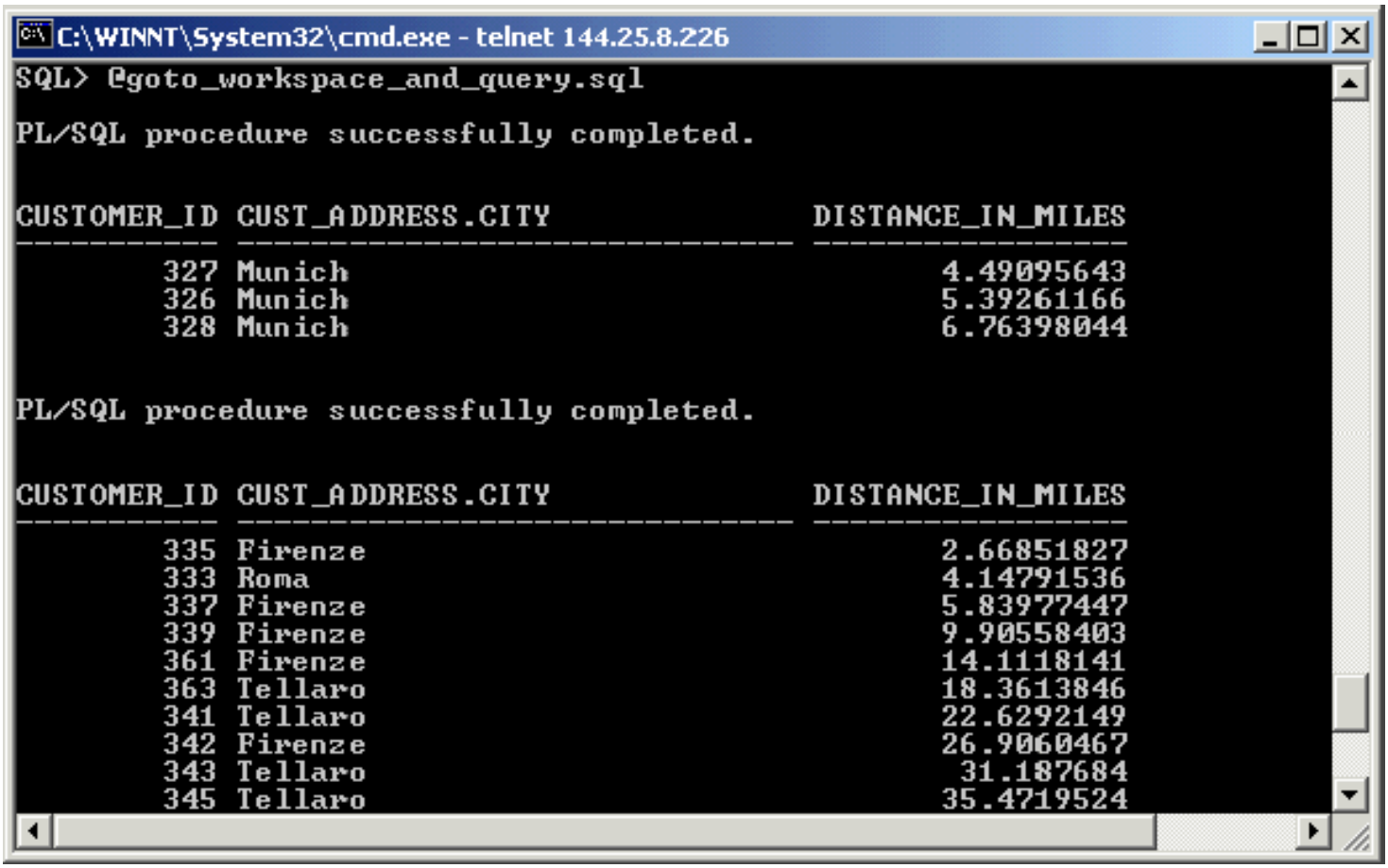
3. Determine which Location is Nearest to the Largest Number of Customers

Back to List

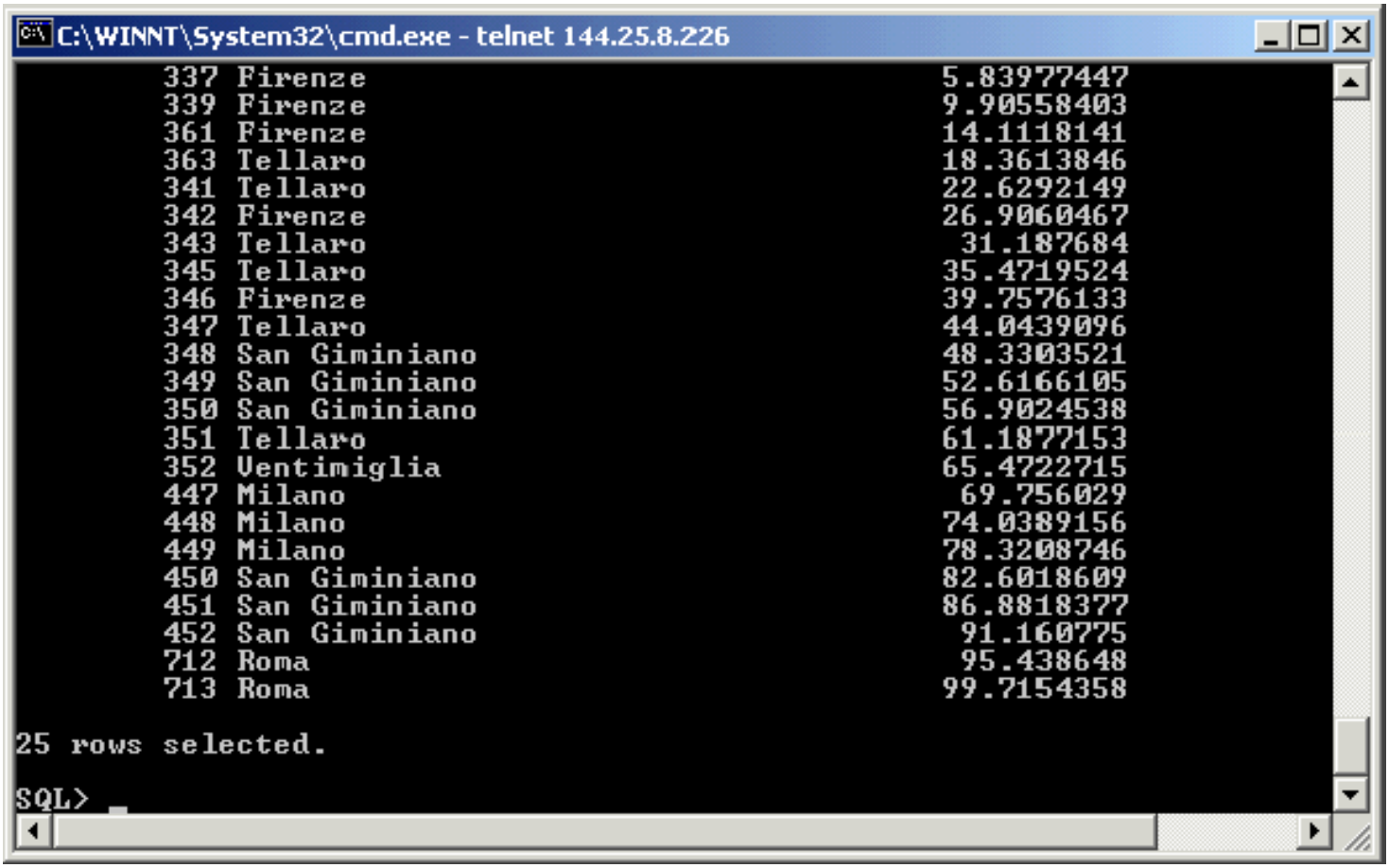
The versioning infrastructure is not visible to the users of the database, and application SQL data manipulation statements (DML) to select, insert, modify, and delete data continue to work in the usual way with version-enabled tables. Users in a workspace automatically see the correct version of the record in which they are interested, that is, the user does not have to keep track of version chains and specify the version of interest.

You will go to each workspace in turn and use existing application SQL to perform a location-based query that reports how many customers are within 100 miles of each site. As it turns out, there are three customers within 100 miles of SITE1 and 25 customers within 100 miles of SITE2. Perform the following steps:

☐ **1.** From a SQL*Plus session logged on to the OE schema, run **@goto_workspace_and_query.sql** .



Scroll down to see the rest of the data.



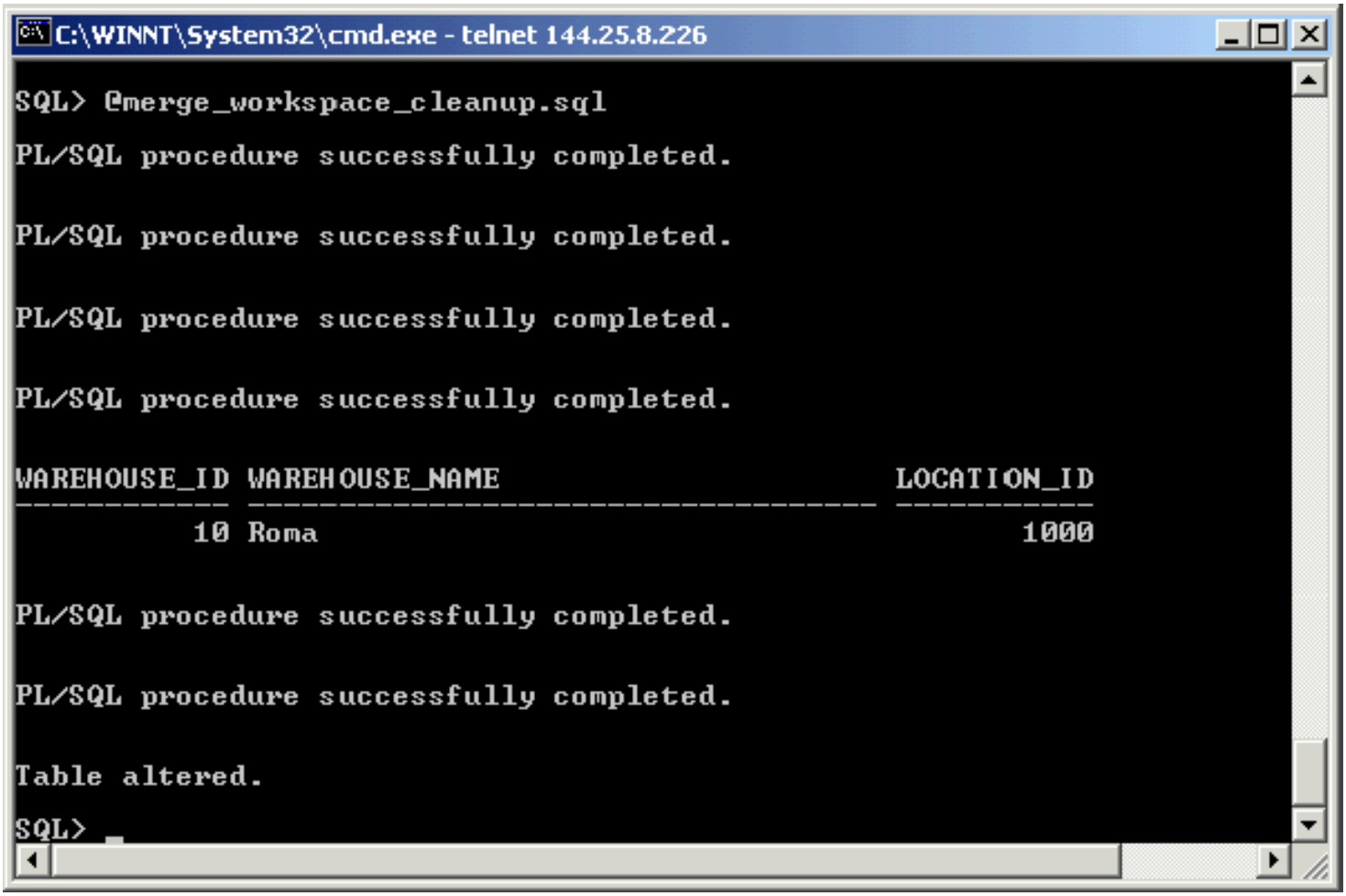
4. Make Chosen Location Data Available to other Users and Clean-up

Back to List

Workspaces can be merged, refreshed or rolled back. Merging a workspace involves applying changes made in a child workspace to its parent workspace. Refreshing a workspace involves applying changes made in the parent workspace to a child workspace. Rolling back a workspace involves deleting changes in the workspace. Users can either delete all changes made since the workspace was created or only changes made after a savepoint. If a row is changed in both the child and parent workspace, a data conflict is created. Conflicts can be checked and resolved at any time. They are automatically detected when a merge or refresh operation is requested.

Since SITE2 has more customers within 100 miles of its location, merge it into workspace LIVE to make SITE2's data accessible to all users. Go to workspace LIVE and "select all from wrehouse" table, SITE2's data now appears. Rollback data for the less desirable SITE1 to remove it, delete the workspace and disable versioning on the warehouse table **.** Perform the following steps:

☐ **1.** From a SQL*Plus session logged on to the OE schema, run **@merge_workspace_cleanup.sql** .



Place the cursor on this icon to hide all screenshots.