Universitat de València

Escola Tècnica Superior d'Enginyeria ETSE-UV

Data Management

Laboratory

Prof. Esther de Ves & Vicente Cerverón

Lab 1: SQL developer

This practice lesson is our first contact with the Oracle SQL Developer working environment, which we will use in our four first sessions:

- Managing connections.
- Viewing, creating and manipulating tables and other Oracle objects.
- Creating and executing SQL queries.
- Other functionalities.

Introduction

BEFORE THE LABORATORY SESSION:

- ✓ Read this document carefully and review database-related subjects in this and previous courses.
- ✓ You may need to remember aspects you studied last year (for example, in this practice lesson you are asked to make several query SQL statements).
- ✓ **You must have SQL Developer installed before the beginning of the laboratory class** and make sure it works properly**.**
- ✓ **Also before class, test the software and connection to the pokemon.uv.es server where the Oracle DBs are located. The exercises in section 1 on Managing Connections must be completed. To help with these exercises, watch the video on First Steps with SQL developer** (*First steps with SQL developer*)

DURING THE LABORATORY SESSION:

- ✓ Pay attention to any further explanations from your teacher.
- ✓ During the session, focus on solving the exercises step by step.
- ✓ If you have a problem with moving forward, let your teacher know.
- ✓ After every three or four exercises you solve, notify your teacher. Don't wait to finish before doing so. If they are busy, continue with the next exercise.

EVALUATION:

- ✓ Show your teacher that you have solved the exercise before leaving the classroom.

Download SQL DEVELOPER at home

We will use SQL Developer in the next four laboratory sessions. This application will be installed on the laboratory computers and in the virtual machine prepared for the course. If you want to also install the application on your own computer, you can do so from the Oracle website. On the download page is a link with documentation. The download page for the version available at the time of writing this guide is:

https://www.oracle.com/tools/downloads/sqldev-downloads.html

This application is free but you must first register with Oracle to be able to download it. A *.zip* file is then downloaded from where you can extract the sqldeveloper folder that contains an executable file that can be used without prior installation.

Inside that folder you will find the executable file, as shown in figure 0, which you can use to launch the program.

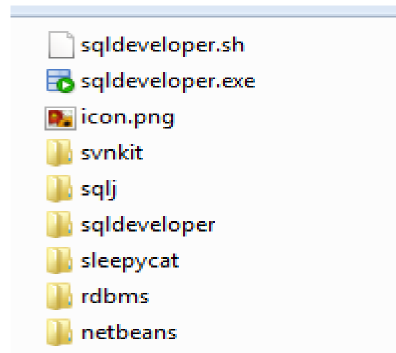


Figure 0: sqldeveloper folder with the executable sqldeveloper.exe file available after unzipping.

l SQL DEVELOPER in the virtual machine

SQL Developer is installed inside the virtual machine (VirtualBox) with Ubuntu OS. Please note the following when downloading and executing the VM:

- Minimum requirements: 6 GB of RAM and 25GB of disk space.
- Downloading the virtual machine: the virtual machine is compressed in a *rar* file that can be downloaded from Dropbox using the link below. The size of the file is roughly 12GB so the process may take a long time:
  https://www.dropbox.com/s/s3hus5j6e476dxv/GCD_BD19_20.rar?dl=0

- Using the VIRTUALBOX virtual machine:
  - Look for the latest version of VirtualBox for your Operating System (OS) before you install it (https://www.virtualbox.org/wiki/Downloads).
  - From the same location, download and install the "VirtualBox XXXXX Oracle VM VirtualBox Extension Pack" (XXXX means the latest version available). It is independent of the OS.

You can run the virtual machine once it is installed. This virtual machine has the sqldeveloper software already installed.

Before working with the virtual machine, make sure you edit the following file:

/home/ubuntu/.sqldeveloper/19.1.0/product.conf

Write the appropriate jdk by changing the corresponding openjdk line to (SetJavaHome):

/usr/lib/jvm/jdk1.8.0.221

To run sqldeveloper, open a terminal window and type the name of the sqldeveloper application in the command line (Video on SQL Developer in the Virtual Machine).

Working with SQL DEVELOPER

Oracle SQL Developer is a development environment that simplifies the creation and management of Oracle databases in traditional and cloud implementations. SQL developer offers:

- a fully comprehensive development tool for your PL/SQL applications,
- a worksheet for executing queries and scripts,
- a DBA console for managing the database,
- a report interface,
- a complete data modelling solution, and
- a migration platform for moving your third-party databases to Oracle.

When SQL DEVELOPER is executed, a main screen with a top menu is displayed. On the left is a menu for navigating between existing objects in the DBs and on the right is an area for displaying information about them (see figure 1):

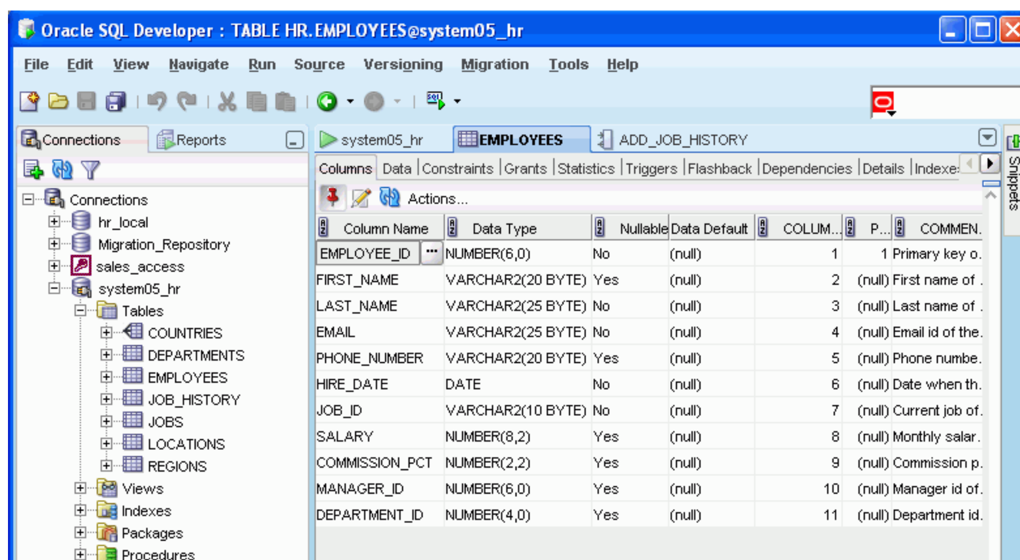


Figure 1: the SQL developer inteface.

At the top are the typical menus of any application plus several features particular to SQL Developer.

On the left-hand side of the window is a panel in which you can display connections and reports about DBs.

- The **connections** panel shows a tree with the created connections. Here, after opening (connection established), you can navigate within the connection's DB objects.
- The **reports** panel enables us to display, in hierarchical fashion, the various reports provided by SQL Developer, such as the list of tables without primary keys and any other report created by the user.

1 Connection management

To work with a Database we first need to learn how to manage the connection to the server where the DB is hosted. A connection in SQL Developer is an object that specifies the information needed to connect to a specific database. You must have at least one connection to work on an Oracle DB. Connections to different DBs, which can be created from this tool, work with several DBs simultaneously.

To create new connections, use the right button to click on the 'Connections' node in the connections browser and select 'New Database Connection' (See Figure 2).

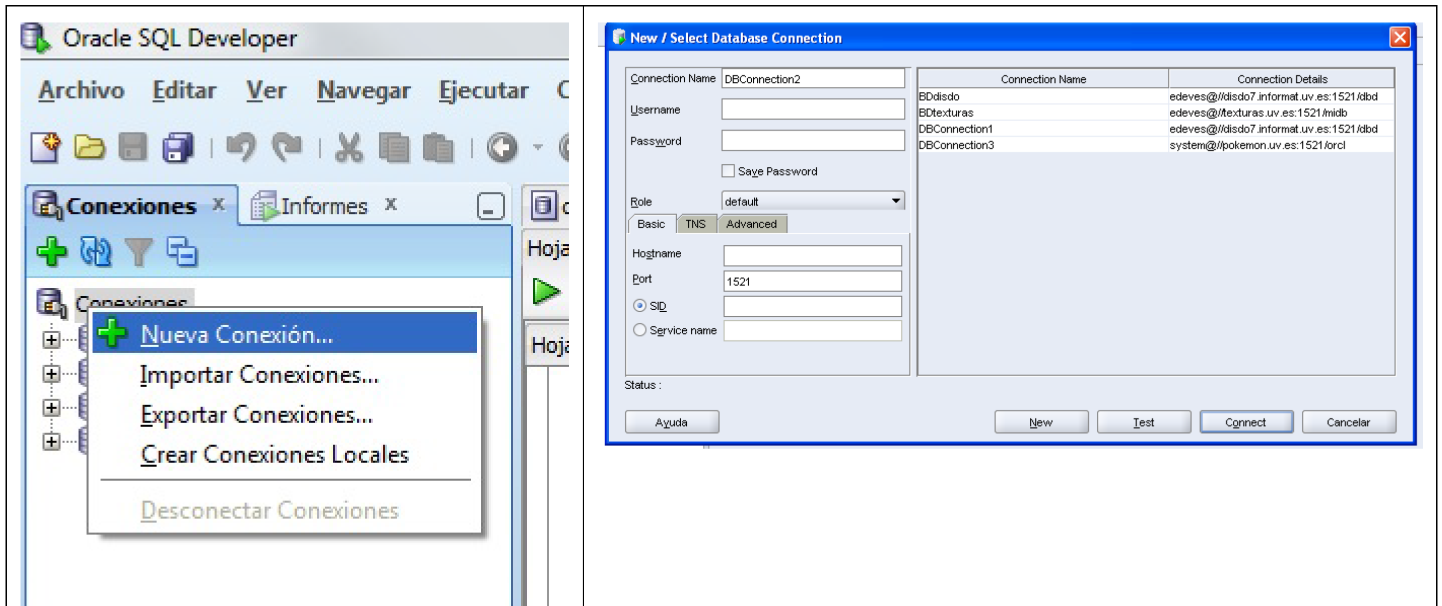


Figure 2: existing connections and how to create a connection (left); properties of a connection (right).

Other operations, which can be performed by clicking on a specific connection, are the collection of statistics (for query optimization) and documentation of a scheme (an HTML document is generated with all existing objects in a given scheme).

1. With the following data, create a connection to the DB:

   Connection Name: any identifying name for the connection you wish to establish.

   Username: username

   Password: password within the BDs.

   Server: pokemon.uv.es

   Port: 1521 (default)

   SID: ORCL (unique identifier for this instance within the defined server)

[Note]: To connect to the pokemon.uv.es server from outside the UV network, you must gain access via the vpn.

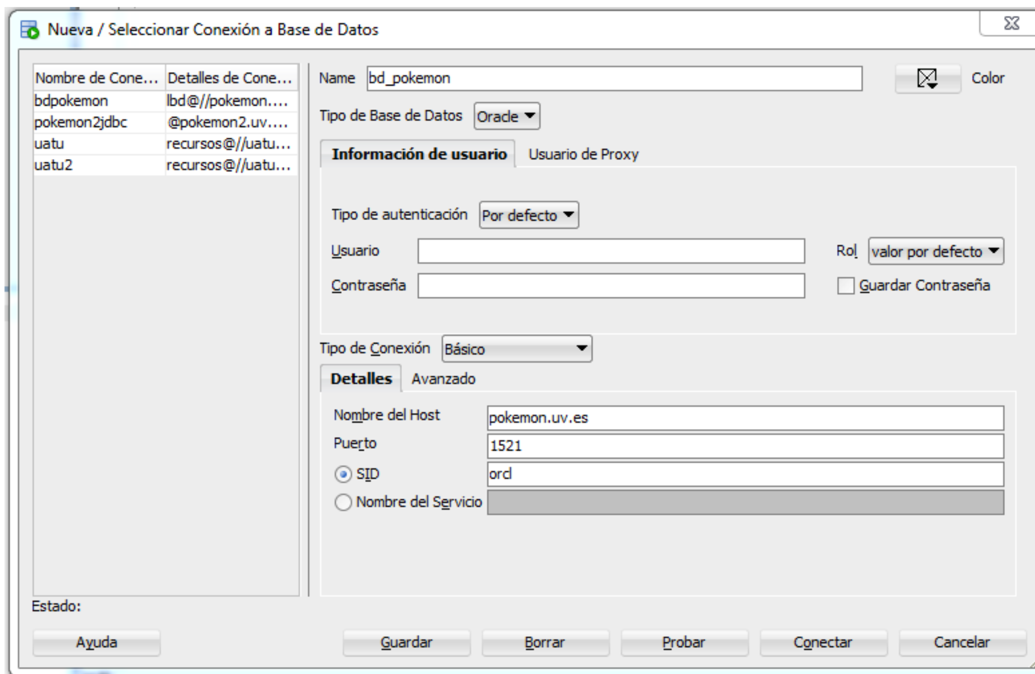


Figure 3: Connection properties.

Exercises: Create a connection.

1. Create the specified connection by using the previous data. You must give the connection a specific name that enables you to identify it within your environment. The **user** and **password** will be provided by the teacher.
2. Check that you can connect to the DB with the connection created (use the "Test" button after filling in the configuration data). Once the test works (no error is detected), connect using the "Connect" button.

NOTE: You must change your username/password after you have established connection by using the following statement executed from an SQL Worksheet (see section 3): ALTER USER ACD0# identified by "newkey";

2 MANAGEMENT OF TABLES, CONSTRAINTS, VIEWS AND INDEXES

How to visualize existing tables

After connecting to a database, you can view the objects created within that database by grouping by user (figure 4). Basic objects in Oracle include Tables, Views, Indexes, Functions, and Other Users, etc. If any objects are selected, other objects of that type are also displayed.

We will review the tables created within the HR (user) scheme.

- Expand the 'Other users' node before expanding the 'HR' node.

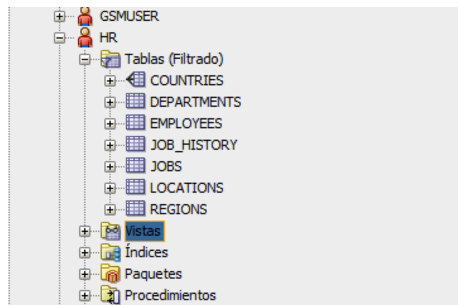
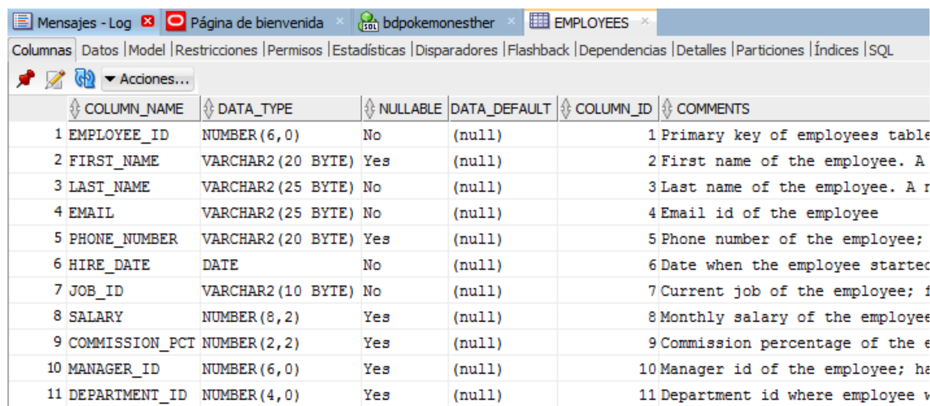


Figure 4: How to see the users schemes.

- Select the EMPLOYEES table: you can visualize all the information included in the table (columns, restrictions, permissions, statistics, and indexes).



	COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
1	EMPLOYEE_ID	NUMBER(6,0)	No	(null)	1	Primary key of employees table
2	FIRST_NAME	VARCHAR2(20 BYTE)	Yes	(null)	2	First name of the employee. A
3	LAST_NAME	VARCHAR2(25 BYTE)	No	(null)	3	Last name of the employee. A r
4	EMAIL	VARCHAR2(25 BYTE)	No	(null)	4	Email id of the employee
5	PHONE_NUMBER	VARCHAR2(20 BYTE)	Yes	(null)	5	Phone number of the employee;
6	HIRE_DATE	DATE	No	(null)	6	Date when the employee startec
7	JOB_ID	VARCHAR2(10 BYTE)	No	(null)	7	Current job of the employee; i
8	SALARY	NUMBER(8,2)	Yes	(null)	8	Monthly salary of the employee
9	COMMISSION_PCT	NUMBER(2,2)	Yes	(null)	9	Commission percentage of the e
10	MANAGER_ID	NUMBER(6,0)	Yes	(null)	10	Manager id of the employee; ha
11	DEPARTMENT_ID	NUMBER(4,0)	Yes	(null)	11	Department id where employee w

Figure 5: Information from a table.

How to create a new table in SQL developer

To create a new table in this environment, expand the 'youruser' node, right-click on the Tables node and select "New table". A window appears, from where you can add new columns to the table while selecting a type of data for each one. You can also modify a table that has already been created (e.g. remove or add columns, remove or add restrictions, etc.). To do so, you must first select the 'Edit table' button (see figure 6).

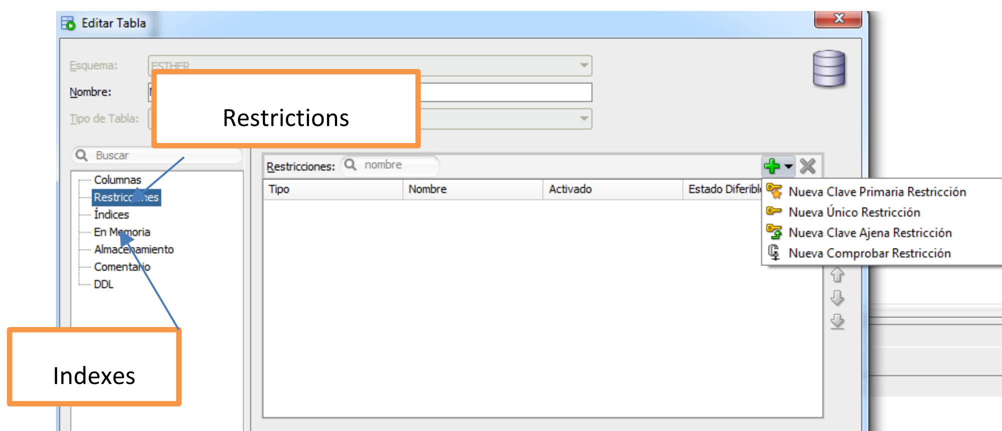


Figure 6: Table edition.

How to display views

To display **views**, you must select the owner scheme of the view you wish to view/edit and select it (same as with the tables).

How to create views

Views, defined from a query in SQL, are used to provide different views of the data depending on the needs of possible users of the DBs. Figure 7 shows the view-creation window. Note that a view has a name and an associated SQL query that can be modified in the text box.

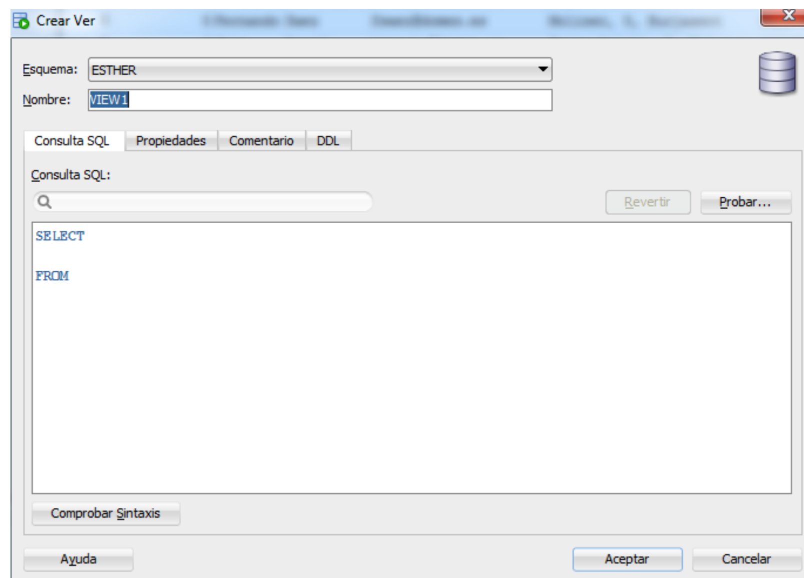


Figure 7: View creation.

How to create a sequence in Oracle

Oracle contains objects, called sequences, which are used mainly for generating attributes (primary key) automatically. They are defined by a name, the beginning of the sequence, the step, and the final value (figure 8).

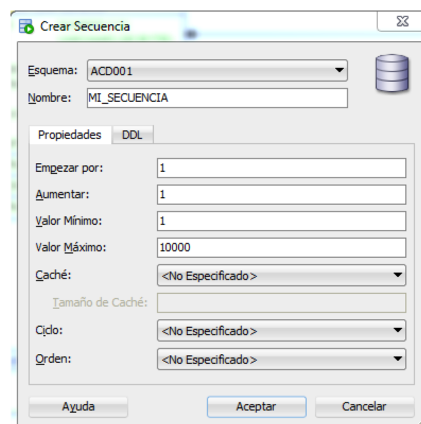


Figure 8: Sequence creation.

VNIVERSITAT DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria ETSE-UV

The sequence can be used to obtain correlative values. To test the sequence, you can use the *currval* (current value) and *nextval* (next value) attribute, which returns the current and next value of the sequence, respectively.

Exercises: Create and edit tables, views, and indexes.

3. Create a first table from your connection. The table, called MI_DEPARTAMENTOS, must contain the following attributes and the following domain. The primary key is the IDDEPARTAMENTO attribute.

COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
IDDEPARTAMENTO	NUMBER(38,0)	No	null	1	null
NOMBRE	VARCHAR2(25 BYTE)	No	null	2	null
MANAGER	NUMBER(38,0)	Yes	NULL	3	null

4. Create the MI_EMPLEADOS table in the same way and with the following structure. The primary key is the IDEMPLEADO attribute.

Columnas | Restricciones | Permisos | Estadísticas | Disparadores | Dependencias | Detalles | Particiones | Índices

COLUMN_NAME	DATA_TYPE	NULLABLE	DATA_DEFAULT	COLUMN_ID	COMMENTS
IDEMPLEADO	NUMBER(38,0)	No	null	1	null
NOMBRE	VARCHAR2(20 BYTE)	No	null	2	null
APELLIDOS	VARCHAR2(30 BYTE)	No	null	3	null
DEPARTAMENTO	NUMBER(38,0)	No	null	4	null
FECHACONTRATO	DATE	Yes	null	5	null
PUESTO	CHAR(8 BYTE)	Yes	null	6	null
NIVELEDUCACION	NUMBER(38,0)	Yes	null	7	null
SUELDO	NUMBER(9,2)	Yes	null	8	null
COMPLEMENTO	NUMBER(9,2)	Yes	null	9	null

5. Once this table is created, we will add two foreign key constraints: one in the ***departmento*** attribute of the MI_EMPLEADOS table with reference to the MI_DEPARTAMENTOS table; and one in the ***manager*** attribute of the MI_DEPARTAMENTOS table with reference to the MI_EMPLEADOS table.
6. Add data to the table. If you double click on the table, the information will appear in the main window. Click on the data tab to enter new records. Insert a couple of new records into each table.

Columnas | Datos | Model | Restricciones | Permisos | Estadísticas | Disparadores | Flashback | Dependencias | Detalles | Particiones | Índices | SQL

Ordenar... | Filtrar:

	IDEMPLEADO	NOMBRE	APELLIDOS	EMAIL	DIRECCION	TELEFONO	TRABAJO	SALARIO
1	1	Juan	Díaz	jdiaz@domes.es	Colon, 3, Valencia	963676767	instructor	1300
2	2	Miguel	Pardo	mpardo@domes.es	Jativa, 13, Valencia	963545454	instructor	1400
3	3	Miguel	Pérez	mperez@domes.es	Guillem, 22, Valencia	963543323	veterinario	2300
4	4	Alicia	San Juan	asanjuan@domes.es	Mayor, 3, Burjassot	962656555	veterinario	2400
5	5	Fernando	Saez	fsaez@domes.es	Moliner, 5, Burjassot	962676667	comercial	950
6	6	Jose	García	jgarcia@domes.es	Poeta Artola, 7, Valencia	963456546	comercial	1200
7	7	Juan	Martínez	jmartinez@domes.es	Moliner,7, Burjassot	962878787	instructor	1500
8	8	Clara	Fernández	clfernandez@gmail.com	Corredera, 8, Liria	639776565	comercial	1400
9	9	Isabel	Cuenca	icuenca@hotmail.com	Corredera, 38, Valencia	967656566	administrativo	1350
10	10	Juana	López	juanlo@gmail.com	Moliner, 3, Burjassot	646897678	comercial	1450
+11	(null)	(null)	(null)	(null)	(null)	(null)	(null)	(null)

Figure 9: You can add data directly from this view of the table (Data). The green record is added. To view this information, place the mouse over the table you wish to view and double click.

7. Create an index in that table on the ***nombre*** attribute. To do so, go to the table and indicate that you wish to create an index. Create a unique index and name it ***idx_dept_name***. Also create non-unique indexes on the foreign keys created.

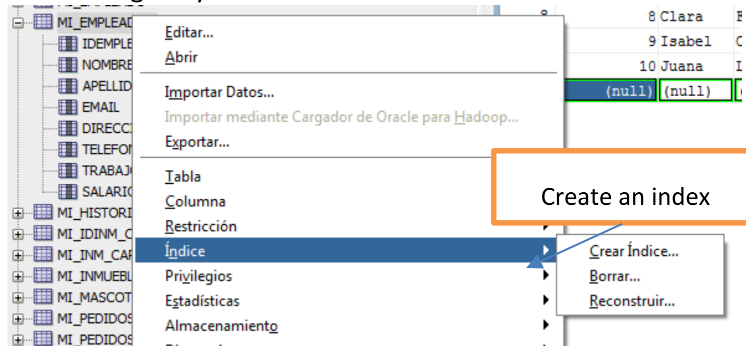


Figure 10: Index creation.

8. Create a view that shows all the fields for employees working in the company after 01/01/2000.
9. Oracle does not contain the AutoNumber Data Type. To simulate this type of data, objects called sequences can be used. To create a sequence, you need a name, an initial value and a final value and to know whether the sequence is ordered and whether it can be re-started at the end. Create a sequence that will help to obtain values for the primary keys. We will see how to use these sequences later.
    a. **dep_seq**
    b. **emp_seq**
    c. **proy_seq**

3. Using the SQL worksheet

SQL Developer can be used to perform basic SQL queries, procedures and functions in Oracle PL-SQL language. We will work with an Oracle tool called 'SQL Worksheet', which enables you to execute SQL statements and SQL scripts.

To run this tool from SQL Developer you can go to the Tools/SQL Worksheet menu (Alt-F10) or use the quick access buttons. It is also accessible from the open connection itself (by clicking on the right mouse button when the cursor is over the connection). Sentences written on a worksheet can be saved to an .sql file and stored on the computer.

The SQL worksheet contains the following icons, which enable you to perform various operations:

View the **query plan**.

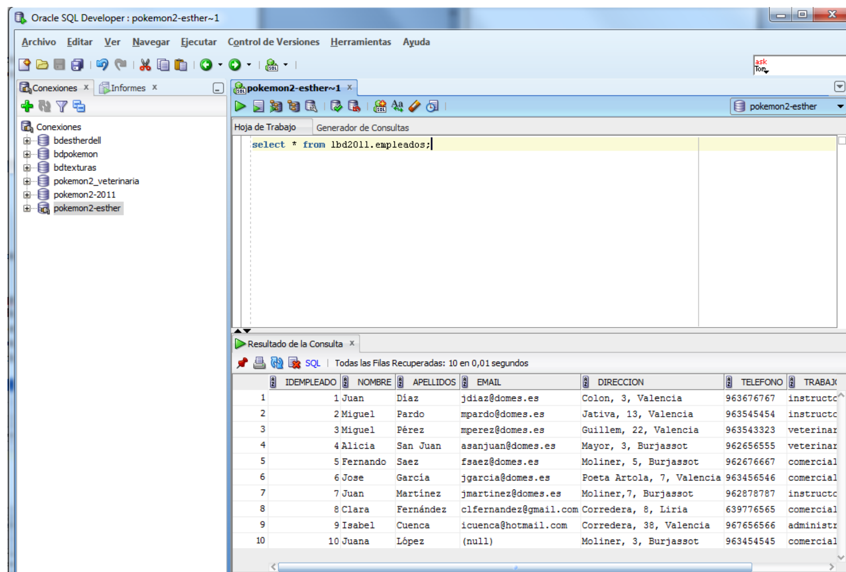


Figure 11: Worksheet from where you can write and execute SQL queries.

These final two options will be used intensively in the next practice lesson.

Running an SQL script in the worksheet

One operation that can be performed with the SQL developer is to load files with sql extensions containing SQL statements.

To do so, you can use the File/Open menu. This opens a worksheet with the file contents. You can modify, execute completely, execute sentence by sentence, etc.

Exercises: SQL scripts

10. Open the file ***01_tables.sql***. Run the script.
11. Open the file ***02_data.sql.*** Run the script.
12. Review the database that has been created (tables, indexes, constraints, sequences, etc.). Write down the following information about the created scheme:
    a. Names of the created tables.
    b. Name, type of constraint and attribute on which the constraint is applied.
    c. Name and index types.
    d. In the 01_tables.sql file, check the use of sequences for generating primary keys automatically.
13. Also review the data that are loaded in the tables.

At this point we now have a scheme of BDs created and populated with data.

Creation and execution of SQL queries on the Worksheet

To run queries on tables created in the DBs, you can use the SQL worksheet. Open a worksheet and write your queries, all of which must end with ';'.

You can run a set of queries (*run script* icon) or a single query (*individual run* icon).

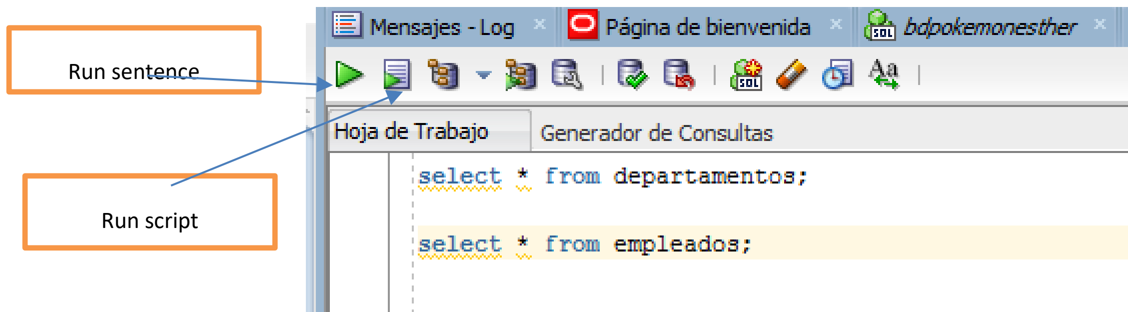


Figure 12: Running SQL statements or scripts.

The result of the query is displayed in the window located at the bottom and in two different formats depending on whether a script is executed (text format) or a statement is executed (table format).

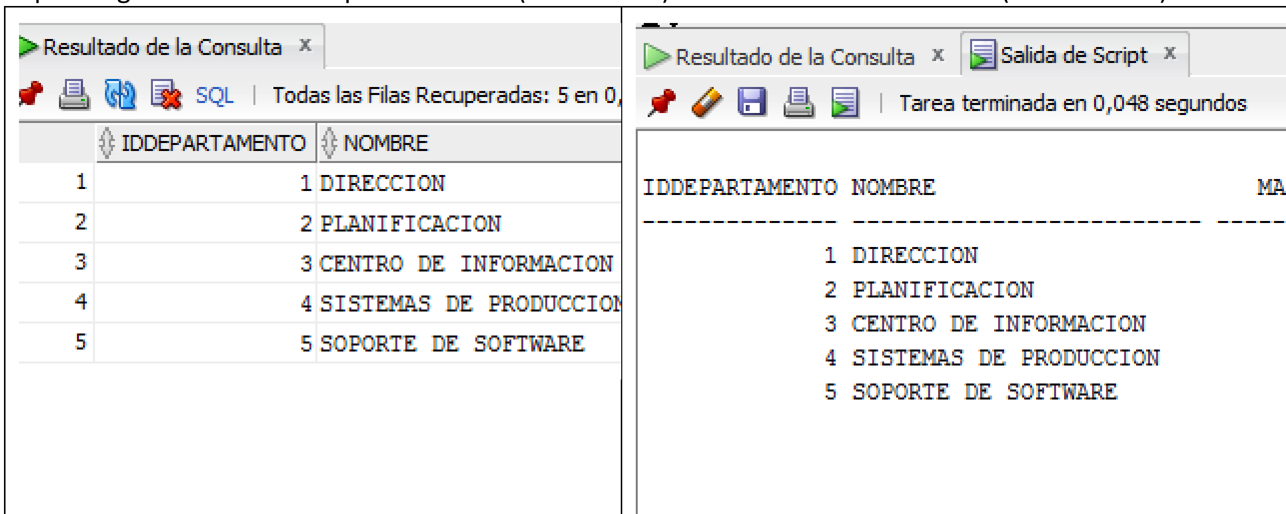


Figure 13: Set of results after statements of scripts (left) and scripts (right) are executed.

Exercises: SQL queries

[Note]: You are expected to be able to write SQL statements of this type. If you can't, review the contents of previous courses.

14. Write a query that returns all the information about employees working in the 'SOPORTE DE SOFTWARE' department.
15. Write a query in SQL that finds employees whose monthly salary (assuming 14 payments) is above 3,000 euros and who have been employed at the company for under 20 years. For this calculation, assume that the salary supplement is part of the salary. **Note:** the *SYSDATE* function can be used to determine the current date.
16. Write a query that finds the number of employees working on each project (the name of the project and the number of employees working on it must be shown).
17. Find the employees whose salary is above the average salary of employees working in the 'CENTRO DE INFORMACION' department.

CONVERTING THE LOGICAL MODEL INTO A CONCEPTUAL MODEL

After you have created the relational scheme, you can easily transform it into a conceptual diagram.

- Select the "Data Modeler" option from the "File" menu.

- From there, choose the “import” option. Select “data dictionary”.
- A window similar to the one below opens up:

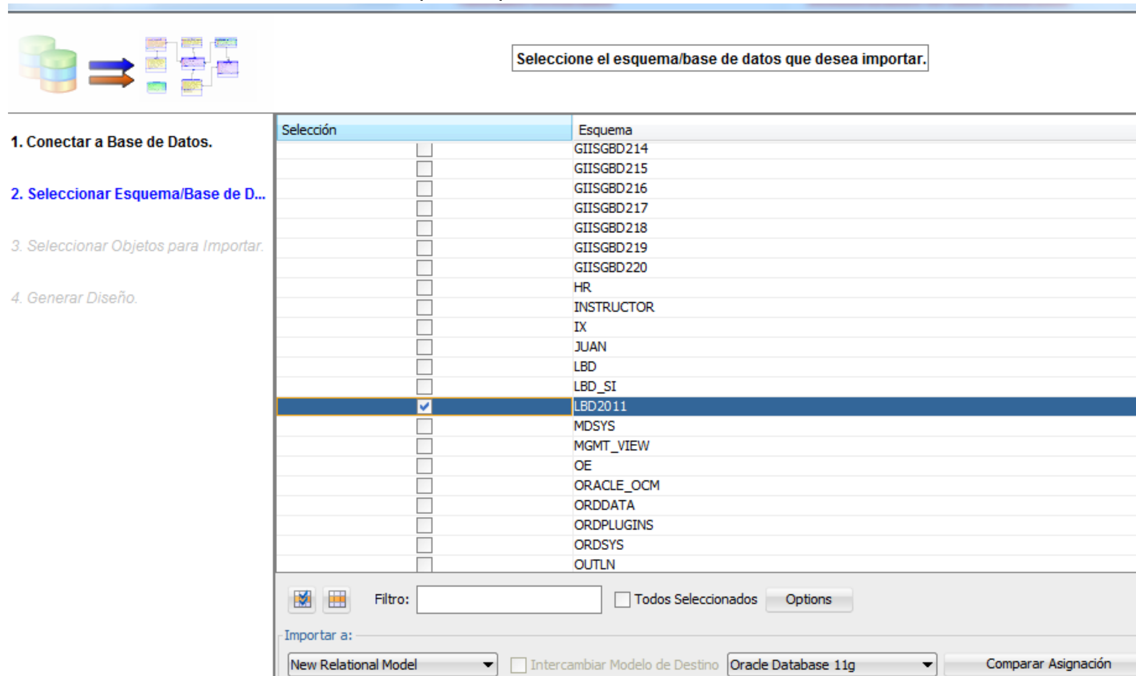


Figure 14: How to obtain the conceptual model from the logical model.

- This window is used to specify the following information:
  - Connect to Database.
  - Select the Database Scheme (User).
  - Select the objects to import.
  - Generate the conceptual design.

Exercises: Logical to conceptual model

18. Generate the conceptual design for the DB you have created in your user.
19. Generate a pdf containing the conceptual design of the company’s DB. To do so, click the right button on the diagram and select print diagram in pdf.

[Note]: WHEN YOU FINISH THIS LABORATORY SESSION, SHOW YOUR TEACHER YOU HAVE SUCCESSFULLY ENDED THE PRACTICE AND THAT YOU HAVE ADEQUATE KNOWLEDGE OF THE SQLDEVELOPER ENVIRONMENT.

VNIVERSITAT DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria ETSE-UV

Lab 2-3: Query optimization in Oracle

PART I. Introduction

The aim of this practice lesson is to familiarize students with the Oracle query-optimization system. The lesson will comprise **two sessions**.

EVALUATION

At the end of each session, you must hand in a script that performs the actions needed to carry out the activity and obtain the necessary information (except initial creation of tables and data loads.

Your text should also contain, in the form of a comment, the answers to the questions asked in each section of the practice.

Your teacher may evaluate your completed exercises during the session, though delivery of the task will be open until the end of each session. Each team must upload the results of their exercises to the *Aula Virtual*:

Practice lesson 2 – Session 1: Exercises 1 to 4.

Practice lesson 2 – Session 2: Exercises 5 to 7.

BEFORE THE LABORATORY SESSIONS

Read the statement carefully and make sure you understand it. **Part II** contains knowledge you will need to apply, references you will need to consult (such as the syntax of some procedures) while you complete the task, and several examples. **Part III** contains the activities you need to complete during the laboratory session.

If necessary, review the concepts you studied in the theoretical part.

If there is anything you don't understand, you can use an online or face-to-face tutorial.

During this practice-preparation phase, take notes to help you solve the laboratory practice exercises more quickly.

To help you prepare for the session, you can also watch the video "Primeros pasos para trabajar con Optimización de Oracle".

DURING THE TWO LABORATORY SESSIONS

Focus on solving the activities using the notes you took earlier. Ask your teacher any new questions that arise. Watch out for possible explanations on the board to solve common problems.

During class and before you leave, show your teacher your progress during the session.

PART II. Important information about query evaluation in Oracle

II.1. Introduction to Oracle Optimizer

An SQL statement can be executed by the DBMS in numerous ways. Each follows a particular execution plan, which is simply an orderly sequence of actions that lead to the results specified in the query. Of all the execution plans, the optimal one is characterized by making minimal use of available resources. The task to discover the best execution plan is performed by the **Query Optimizer**, a common component in DBMS.

The query optimizer determines the most efficient way to execute an SQL statement after considering numerous factors related to the objects involved and the query's specific conditions. It generates an **execution plan** for each query that describes the execution method via several steps that retrieve records from the database or prepare them for use in the next step.

The optimizer in the Oracle DBMS has changed considerably since its first versions. Initially, it used a series of rules to find the best plan (RBO, i.e. Rule Based Optimizer). The main problem with an optimizer with these characteristics is that it applies the same rules to all objects regardless of their conditions, which can lead to errors (i.e. failure to find the best plan). From version 7 onwards, the optimizer operation was improved by introducing a calculation mechanism based on costs and statistics (CBO, i.e. Cost-Based Optimizer). Its main advantage over the previous version is that it took into account the actual content and distribution of data (statistics on objects) to calculate the execution costs of several plans and select the lowest cost. These plans are based on screening criteria that limit the possibilities (which grow exponentially depending on the number of objects involved in the consultation).

To understand how the query optimizer works, therefore, we need to know how Oracle stores and manages the statistics on the objects and to understand the alternatives that can be used to execute a plan. In this practice lesson, which covers two sessions, we will work on both of these aspects.

To understand what is discussed below we must remember these simple ideas:

The composition of the DB	*Where they are obtained from*	*What is used*
Tables (and their columns), indexes, etc.	*Statistics*	*The optimizer*

II.2. STATISTICS MANAGEMENT

BASIC DESCRIPTION OF STATISTICS

The statistics used by the optimizer are a collection of data that describe the database and its objects, and enable the best execution plan to be calculated for each SQL statement. The statistics are stored in the database's data dictionary (system catalog), and are accessible through views such as USER_TABLES, USER_TAB_COLUMNS, USER_INDEXES, USER_TAB_STATISTICS, USER_IND_STATISTICS and USER_TAB_COL_STATISTICS. Using SELECT type queries on these tables, we can therefore consult the statistical information that is available to the optimizer.

Details of the columns available in each view can be found in the Oracle documentation:

USER_TABLES *https://docs.oracle.com/cd/B19306_01/server.102/b14237/statviews_4473.htm#REFRN26286*
USERT_TAB_COLUMNS *https://docs.oracle.com/cd/B19306_01/server.102/b14237/statviews_4462.htm#REFRN26277*
USER_INDEXES *https://docs.oracle.com/cd/B19306_01/server.102/b14237/statviews_4334.htm#i1633820*

The statistics corresponding to the tables contain information such as the number of rows, the number of data blocks (logical storage) used by the table, and average length of the records. In relation to the columns, the statistics include information about the number of different values it contains, as well as the maximum and minimum values of each column. The optimizer uses the table statistics to estimate the cost of accessing the data and the column statistics to estimate the number of records that will be retrieved at each step of the execution plan.

Although these basic statistics are an important starting point for the optimizer, they do not provide all the information needed for a correct calculation. We also need to know the nature of the information, including biases, distortions, the distributions of values in a column, and the existence of any correlation between columns. For this purpose, Oracle provides a set of additional statistics:

- Histograms: these contain information about the distribution of data in a column. If this statistic does not exist, Oracle assumes a uniform data distribution.
  - Frequency histograms: these are used when the number of different values in a column is equal or smaller than 254.
  - Histograms of balanced height: these are used when the number of different values in a column is greater than 254.
- Column groups: these contain information about columns that have some kind of correlation (e.g. province/country). They make sense when the columns of the same table are used together in the WHERE clause of a query since they enable you to estimate the cardinality of the resulting set more accurately.
- Expressions: Oracle also enables the creation of statistics on expressions, including functions, in order to improve cost estimation. For example, if the UPPER function is frequently used on a column, it would be useful to extend the statistics to include this expression.

Extended statistics are captured manually using the DBMS_STATS.CREATE_EXTENDED_STATS procedure and can be consulted in the views of the USER_TAB_COL_STATISTICS and USER_STAT_EXTENSIONS data dictionary.

Oracle also maintains statistical information on the use of indexes. This information is essential for determining whether it is a good idea in an execution plan to use data access through indexes or directly on the table. The statistical information from the indexes includes the number of values (different keys), the depth of the index (height), the number of blocks in the leaf nodes, and the clustering factor.

METHODS FOR GATHERING STATISTICS

Object statistics in the Oracle database are obtained through the DBMS_STATS PL / SQL package. This contains over 50 procedures for collecting statistics but the most important ones are those of the form GATHER _ * _ STATS, since they collect information about tables, columns and indexes. To be able to execute the procedures for acquiring statistics on the objects, you need to be their owner (or have the ANALYZE ANY system privilege).

This same package has a procedure that automates the process of obtaining statistics during a time window for those objects that need it. If it is to work properly, certain parameters related to the way information is collected must be configured.

In addition to the automatic procedure, statistics can be collected manually when needed. Below we will discuss in greater detail the main manual procedures and how to use them.

To understand quickly what each procedure does, remember this summary:

GATHER_*_STATS	*Update statistics with the current information about an object type **
DELETE_*_STATES	*Drop statistics from objects type **
*Where * can be TABLE, INDEX, SCHEMA, etc.*	

In this practice lesson you must also bear in mind:

- The behaviour when no statistics are available yet. In a table that has just been created and filled in, and from which statistics have not yet been collected, if we do nothing, to decide on an execution plan, a sampling is carried out to estimate certain values, which will normally be an approximation to the current tables.
- With automatic statistics, on the other hand, the statistics are automatically recomputed every night (if an exercise is left halfway through and resumed the next day, we will see that statistics have already been calculated in the tables).

GATHER_TABLE_STATS

The procedure is as follows:

```
DBMS_STATS.GATHER_TABLE_STATS (
   ownname          VARCHAR2,
   tabname          VARCHAR2,
   partname         VARCHAR2 DEFAULT NULL,
   estimate_percent NUMBER   DEFAULT to_estimate_percent_type
                                              (get_param('ESTIMATE_PERCENT')),
   block_sample     BOOLEAN  DEFAULT FALSE,
   method_opt       VARCHAR2 DEFAULT get_param('METHOD_OPT'),
   degree           NUMBER   DEFAULT to_degree_type(get_param('DEGREE')),
   granularity      VARCHAR2 DEFAULT GET_PARAM('GRANULARITY'),
   cascade          BOOLEAN  DEFAULT to_cascade_type(get_param('CASCADE')),
   stattab          VARCHAR2 DEFAULT NULL,
   statid           VARCHAR2 DEFAULT NULL,
   statown          VARCHAR2 DEFAULT NULL,
   no_invalidate    BOOLEAN  DEFAULT  to_no_invalidate_type (
                                   get_param('NO_INVALIDATE')),
   force            BOOLEAN DEFAULT FALSE);
```

The mandatory parameters are the owner of the table and the name of the table, while the rest are optional and in some cases taken from the database configuration. Also important are the CASCADE parameter, which collects statistics in the indexes associated with the table, and the FORCE parameter, which collects statistics even when the table is locked.

Use example:

```
Exec
dbms_stats.gather_table_stats(ownname=>'ACD050',tabname=>'ACTORES',cascade=>true,force=>t
rue);
```

```
begin

dbms_stats.gather_table_stats(ownname=>'ACD050',tabname=>'ACTORES',cascade=>true,
force=>true);

end;

/
```

DELETE_TABLE_STATS

The procedure is as follows:

```
DBMS_STATS.DELETE_TABLE_STATS (
   ownname          VARCHAR2,
   tabname          VARCHAR2,
   partname         VARCHAR2 DEFAULT NULL,
   stattab          VARCHAR2 DEFAULT NULL,
   statid           VARCHAR2 DEFAULT NULL,
   cascade_parts    BOOLEAN  DEFAULT TRUE,
   cascade_columns  BOOLEAN  DEFAULT TRUE,
   cascade_indexes  BOOLEAN  DEFAULT TRUE,
   statown          VARCHAR2 DEFAULT NULL,
   no_invalidate    BOOLEAN  DEFAULT to_no_invalidate_type (
                                 get_param('NO_INVALIDATE')),
   force            BOOLEAN DEFAULT FALSE);
```

The mandatory parameters are the owner of the table and the name of the table, while the rest are optional. Also important are the CASCADE_COLUMNS and CASCADE_INDEXES parameter, which enable statistics to be deleted from the columns and indexes associated with the table, and the FORCE parameter, which deletes statistics even when the table is locked.

Example of use:

```
exec dbms_stats.delete_table_stats(ownname=>'ACD050',tabname=>'ACTORES');
```

DELETE_COLUMN_STATS

The procedure is as follows:

```
DBMS_STATS.DELETE_COLUMN_STATS (
   ownname       VARCHAR2,
   tabname       VARCHAR2,
   colname       VARCHAR2,
   partname      VARCHAR2 DEFAULT NULL,
   stattab       VARCHAR2 DEFAULT NULL,
   statid        VARCHAR2 DEFAULT NULL,
   cascade_parts BOOLEAN  DEFAULT TRUE,
   statown       VARCHAR2 DEFAULT NULL,
   no_invalidate BOOLEAN  DEFAULT to_no_invalidate_type (
                                 get_param('NO_INVALIDATE')),
   force         BOOLEAN DEFAULT FALSE);
```

The mandatory parameters are the owner of the table, the name of the table and the name of the column, while the rest are optional. The FORCE parameter enables statistics corresponding to the column to be deleted even if it is locked.

Example of use:

```
exec
dbms_stats.delete_column_stats(ownname=>'ACD050',tabname=>'ACTORES',colname=>'SEXO');
```

GATHER_INDEX_STATS

The procedure has the following syntax:

```
DBMS_STATS.GATHER_INDEX_STATS (
   ownname          VARCHAR2,
   indname          VARCHAR2,
   partname         VARCHAR2 DEFAULT NULL,
   estimate_percent NUMBER   DEFAULT to_estimate_percent_type
                                        (GET_PARAM('ESTIMATE_PERCENT')),
   stattab          VARCHAR2 DEFAULT NULL,
   statid           VARCHAR2 DEFAULT NULL,
   statown          VARCHAR2 DEFAULT NULL,
   degree           NUMBER   DEFAULT to_degree_type(get_param('DEGREE')),
   granularity      VARCHAR2 DEFAULT GET_PARAM('GRANULARITY'),
   no_invalidate    BOOLEAN  DEFAULT to_no_invalidate_type
                                       (GET_PARAM('NO_INVALIDATE')),
   force            BOOLEAN DEFAULT FALSE);
```

The mandatory parameters are the owner of the index and the name of the index, while the rest are optional and again in some cases taken from the database configuration. The FORCE parameter enables index statistics to be collected even if index is locked.

Example of use:

```
exec dbms_stats.gather_index_stats(ownname=>'ACD050',indname=>'IDX_SEXO',force =>true);
```

DELETE_INDEX_STATS

The procedure is as follows:

```
DBMS_STATS.DELETE_INDEX_STATS (
   ownname          VARCHAR2,
   indname          VARCHAR2,
   partname         VARCHAR2 DEFAULT NULL,
   stattab          VARCHAR2 DEFAULT NULL,
   statid           VARCHAR2 DEFAULT NULL,
   cascade_parts    BOOLEAN  DEFAULT TRUE,
   statown          VARCHAR2 DEFAULT NULL,
   no_invalidate    BOOLEAN  DEFAULT to_no_invalidate_type (
                                 get_param('NO_INVALIDATE')),
   force            BOOLEAN DEFAULT FALSE);
```

The mandatory parameters are the owner of the index and the name of the index. The FORCE parameter allows statistics to be deleted from the index even if it is locked.

Example of use:

```
exec dbms_stats.delete_index_stats(ownname=>'ACD050',indname=>'IDX_SEXO',force=>true);
```

GATHER_SCHEMA_STATS

The procedure is as follows:

```
DBMS_STATS.GATHER_SCHEMA_STATS (
   ownname          VARCHAR2,
   estimate_percent NUMBER   DEFAULT to_estimate_percent_type
                                               (get_param('ESTIMATE_PERCENT')),
   block_sample     BOOLEAN  DEFAULT FALSE,
   method_opt       VARCHAR2 DEFAULT get_param('METHOD_OPT'),
   degree           NUMBER   DEFAULT to_degree_type(get_param('DEGREE')),
   granularity      VARCHAR2 DEFAULT GET_PARAM('GRANULARITY'),
   cascade          BOOLEAN  DEFAULT to_cascade_type(get_param('CASCADE')),
   stattab          VARCHAR2 DEFAULT NULL,
   statid           VARCHAR2 DEFAULT NULL,
   options          VARCHAR2 DEFAULT 'GATHER',
   objlist          OUT      ObjectTab,
   statown          VARCHAR2 DEFAULT NULL,
   no_invalidate    BOOLEAN  DEFAULT to_no_invalidate_type (
                                     get_param('NO_INVALIDATE')),
  force             BOOLEAN DEFAULT FALSE);
```

The only mandatory parameter is the name of the scheme from which the statistics are to be taken. The CASCADE option allows you to collect additional statistics from the indexes (which is not done by default), while the FORCE option avoids any blocks that may exist in the objects.

Example of use:

```
exec
dbms_stats.gather_schema_stats(ownname=>'ACD050',cascade=>true,force=>true);
```

DELETE_SCHEMA_STATS

The procedure is as follows:

```
DBMS_STATS.DELETE_SCHEMA_STATS (
   ownname          VARCHAR2,
   stattab          VARCHAR2 DEFAULT NULL,
   statid           VARCHAR2 DEFAULT NULL,
   statown          VARCHAR2 DEFAULT NULL,
   no_invalidate    BOOLEAN DEFAULT to_no_invalidate_type (
                                     get_param('NO_INVALIDATE')),
   force            BOOLEAN DEFAULT FALSE);
```

The only mandatory parameter is the name of the scheme from which you wish to delete the statistics. The FORCE option avoids any blocks that may exist in the objects.

Example of use:

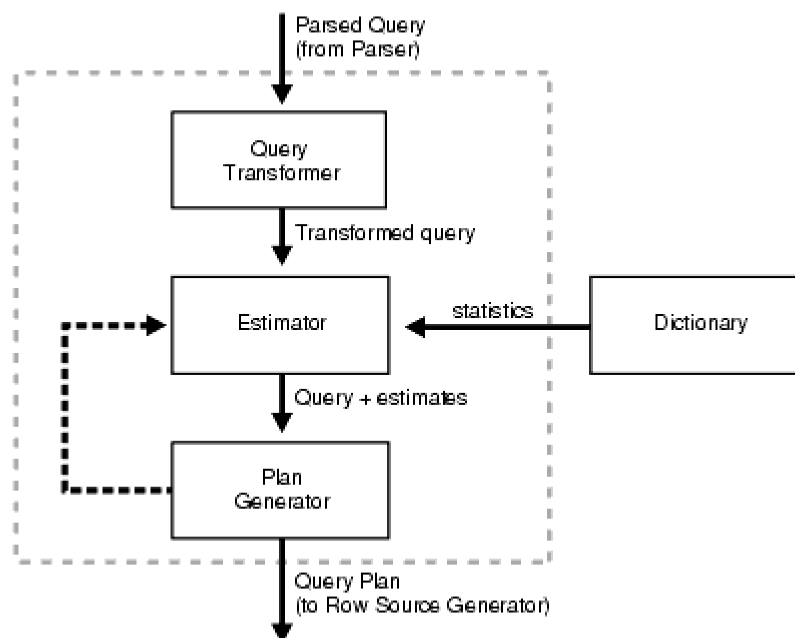
```
exec dbms_stats.delete_schema_stats(ownname=>'ACD050',force=>true);
```

II.3 EXECUTION PLANS

Basic description of execution plans

An execution plan shows in detail the steps needed to execute an SQL statement. These steps are expressed in the form of operations based on data that consume and produce records. The order of operations and the way to implement them are decided by the query optimizer, which uses a combination of query transformations and physical optimization techniques (data access) to do so.

The components of the Oracle query optimizer are:



The plan generator is responsible for trying various alternatives for executing a specific query and selecting the one with the lowest cost. Many alternatives exist because there are several combinations of data access paths, meeting methods and orders of meetings that would generate the same end result but with different costs.

Based on the statistics it has of the objects involved in a query, the Oracle optimizer will select the best execution plan for it. When designing queries that involve very heavy objects (i.e. with a lot of data), it is therefore mandatory to keep the statistics updated so that the optimizer will find the best plan (and consume the minimum possible resources). The execution plan should also be reviewed to determine whether it can be improved by physically re-organizing the data (indexes, partitions, etc.). This process is called SQL Query Tuning.

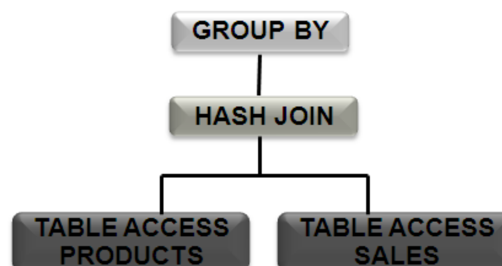
The basic elements for obtaining and analysing query execution plans in Oracle are described below.

Calculation of the execution plan

To facilitate understanding by the user, in Oracle a query's execution plan is shown in tabular form rather than tree-like form (the usual way to represent the plans but which cannot be implemented in a text-type interface such as that provided by Oracle database query tools, type SQL * Plus). It looks similar to this:

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop
0	SELECT STATEMENT				1140 (100)			
1	HASH GROUP BY		4	80	1140 (45)	00:00:14		
* 2	HASH JOIN		489K	9555K	792 (21)	00:00:10		
3	TABLE ACCESS FULL	PRODUCTS	767	8437	10 (0)	00:00:01		
4	PARTITION RANGE ALL		489K	4300K	741 (17)	00:00:09	1	16
5	TABLE ACCESS FULL	SALES	489K	4300K	741 (17)	00:00:09	1	16

This execution plan would correspond to an execution tree like this one:



The execution plan shows information about the operations carried out, the objects involved, an estimation of the records handled by each operation, the memory required for each operation, the cost associated with each operation, and the estimated execution time, etc. These correspond to the columns Operation, Name, Rows, Bytes, Cost and Time, respectively. The cost measure is an internal measure of the database; it only makes sense to use it when comparing execution plans.

The most complex aspect to understand perhaps lies in operations. Several types of operations can be performed in the records. These can be divided into:

- **Access methods:**
  - **Full Table Scan**. This reads all the records in a table and filters those that satisfy the criteria of the WHERE clause.
  - **Table Access by ROWID**. This specifies the exact position where a record resides. Oracle first obtains ROWID through an index or by filtering the WHERE clause and then accessing the data record by record.
  - **Index Unique Scan**. This retrieves only one record after traversing a single index.
  - **Index Range Scan**. This accesses adjacent index entries and then locates the records corresponding to the recovered ROWIDs.
  - **Index Range Scan Descending**. This is the same as the previous case but is applied when the index is traversed in reverse due to an ORDER BY … DESC.

- **Index Skip Scan**. This is used when all columns present in the index except the first one appear.
- **Full Index Scan.** Despite what its name suggests, this does not read all the index blocks. Here, Oracle processes all index sheet blocks until it finds the first one it needs.
- **Fast Full Index Scan**. This is used as an alternative to the full table scan when the index contains all the columns needed for the query and at least one column of the index has the NOT NULL restriction.
- **Index Join**. This is an index meeting of the same table that collectively contains all the columns needed for the query.
- **Bitmap Index**. This uses a bitmap for key values, and each position is mapped into a ROWID.

- **Join Methods**
  - **Hash join.** This is used for very large tables of data. The optimizer uses the smallest table to build a hash table, based on the meeting key, and then scans the largest table using the same hash function to look for the matching values.
  - **Nested Loop Join**. This applies to small data sets. For each record in the first table, all records in the second table are accessed.
  - **Sort Merge Join.** This is useful when the join condition of the two tables is an inequality. In such cases, it works better than the Nested Loop Join. Ordering is applied to both sets on the key of the join and then proceeds to the join.
  - **Cartesian Join.** With this, all tuples of one table are combined with those of the other table. It only applies to small data sets or when the query requires it because there are no meeting criteria.

The join methods also involve an order in the performance of the operation, which is shown in the table in the form of indentation. This means that objects with more indentation (more bleeding) come together first. From there, the result meets the next indented element, and so on until the last operation is reached. Depending on the method used, the order of the join is important since, depending on the cardinality of the join it may entail different costs.

Execution plans are stored in a table in the database data dictionary called PLAN_TABLE. This table is usually created in the database and prepared for each user to use independently. If the table is not created, it can be created by running the *utlxplan.sql* script located in the $ ORACLE_HOME / rdbms / admin directory where the database is installed. The script can also be used to customize the plan table and avoid concurrent access problems between different sessions and/or users.

To obtain the execution plan for a query, the EXPLAIN PLAN command, which populates the PLAN_TABLE table, is used. Visualization of the plan as a query on the table is rather tedious and complicated, so the DBMS_XPLAN package is usually used. This package is responsible for providing the tabular view, properly formatted for better understanding as shown at the beginning of this section.

The EXPLAIN PLAN command is:

```
EXPLAIN PLAN FOR
  <query>;
```

For instance:

```
EXPLAIN PLAN FOR
     SELECT last_name FROM employees;
```

This command stores the execution plan in the PLAN_TABLE table. However, it is possible to store the plan in a custom table (a copy of the PLAN_TABLE table) using this syntax:

EXPLAIN PLAN

```
EXPLAIN PLAN
    INTO <table>
  FOR <query>;
```

For example:

```
EXPLAIN PLAN
     INTO my_plan_table
   FOR
     SELECT last_name FROM employees;
```

Once the plan creation and storage process has been executed, it can be consulted. To do so, the DISPLAY procedure of the DBMS_XPLAN package, which consults the last SQL statement analysed and stored in the plan table, is used. The syntax for the procedure is:

```
DBMS_XPLAN.DISPLAY(
   table_name     IN  VARCHAR2  DEFAULT 'PLAN_TABLE',
   statement_id   IN  VARCHAR2  DEFAULT  NULL,
   format         IN  VARCHAR2  DEFAULT  'TYPICAL',
   filter_preds   IN  VARCHAR2 DEFAULT NULL);
```

As we can see, all parameters are optional. However, if a custom plan table is being used, it must be indicated in the TABLE_NAME parameter. Similarly, the FORMAT parameter indicates the amount of information to be displayed. Any of the following values can be specified:

- BASIC. This shows only the minimum details of the plan, which are the operation and the object on which the action is performed.
- TYPICAL. This contains the most relevant information for fully understanding the plan, which includes estimates of records and memory, costs, temporary estimates, and degrees of parallelization.
- SERIAL. This contains the same information as TYPICAL except for parallelization.
- ALL. This shows all available information.

Combinations of parameters can also be specified to be displayed in this field, including 'ALL –PROJECTION –NOTE', 'TYPICAL PROJECTION', '–BYTES –COST –PREDICATE'.

The procedure is

```
SELECT plan_table_output
   FROM
        table(DBMS_XPLAN.DISPLAY('PLAN_TABLE',
        NULL,'TYPICAL'));
```

or

```
   SELECT * FROM table(DBMS_XPLAN.DISPLAY);
```

With these basic concepts on execution plans and ensuring that statistics on work objects are properly updated, SQL queries can be analysed for optimal execution.

About the query execution

Oracle provides several types of information about the form and result of executing a query. EXPLAIN PLAN enables us to obtain the execution plan a query would use without having to launch it. Execution plans are an important component of the information obtained, since they identify the procedure the manager will follow to obtain the desired result.

As well as the execution plans, two other types of information reveal the conditions of the database instance for accessing the information (basically statistics on the I/O operations to solve the query), also known as trace, and the execution time used. In this case, it involves obtaining information from a query that is executed. One possibility we have is to obtain the execution plan.

The trace of a query enables us to view the execution plan and the statistics and can be obtained by activating an option in the query interface (*autotrace* parameter). Several forms and conditions of activation exist:

```
SET AUTOTRACE ON EXPLAIN
     The AUTOTRACE report shows only the optimizer execution
     path.

SET AUTOTRACE ON STATISTICS
     The AUTOTRACE report shows only the SQL statement execution
     statistics.

SET AUTOTRACE ON
     The AUTOTRACE report includes both the optimizer execution
     path and the SQL statement execution statistics.
```

```
SET AUTOTRACE TRACEONLY
     Like SET AUTOTRACE ON, but suppresses the printing of the
     user's query output, if any.
```

The time used to execute the consultation can also be visualised by activating the timing parameter as follows:

```
SET TIMING ON
```

II.4 BIBLIOGRAPHY

Understanding Optimizer Statistics.
http://www.oracle.com/technetwork/database/focus-areas/bi-datawarehousing/twp-optimizer-stats-concepts-110711-1354477.pdf

Best practices for gathering optimizer statistics.
http://www.oracle.com/technetwork/database/focus-areas/bi-datawarehousing/twp-bp-optimizer-stats-04042012-1577139.pdf

DBMS_STATS package.
http://docs.oracle.com/cd/B19306_01/appdev.102/b14258/d_stats.htm

The Oracle Query Optimizer.
http://docs.oracle.com/cd/B28359_01/server.111/b28274/optimops.htm

The Oracle Optimizer. Explain the Explain Plan.
http://www.oracle.com/technetwork/database/focus-areas/bi-datawarehousing/twp-explain-the-explain-plan-052011-393674.pdf

Using EXPLAIN PLAN.
http://docs.oracle.com/cd/B28359_01/server.111/b28274/ex_plan.htm

DBMS_XPLAN package.
http://docs.oracle.com/cd/B19306_01/appdev.102/b14258/d_xplan.htm

Part II. Laboratory Activities

This laboratory session will be conducted with the Oracle 19g database installed on the **pokemon.uv.es** server, for which all students have access credentials.

TABLES BASE CREATION

To conduct this practice lesson we need to create three tables in the user scheme, as described below.

Table: ACTORES

COLUMN	TYPE	PK	NN	UNI	CHK	FK	COMMENT
OID	NUMBER(10,0)	✓					
NOM	CHAR(100)		✓				
SEXO	CHAR(1)		✓				

Table: PELICULAS

COLUMN	TYPE	PK	NN	UNI	CHK	FK	COMMENT
OID	NUMBER(10,0)	✓					
TITULO	CHAR(200)		✓				
ANYO	NUMBER(4,0)		✓				

Table: ACTUACION

COLUMN	TYPE	PK	NN	UNI	CHK	FK	COMMENT
ACTOR	NUMBER(10,0)	✓					
PELI	NUMBER(10,0)	✓					
PAPEL	CHAR(100)	✓					

The script for generating these tables and the statement are provided in the *crea_tablas.sql* file. **The first thing you will have to do (before exercise 1), therefore, is to run the script.**

Note: *Loading the .sql completely in the worksheet and running the script may give you a problem. Therefore, use this other, more efficient, way of executing an sql script (such as create_tables.sql, and data2.sql) by executing this command from the worksheet (adding your path and the corresponding .sql file name):*

@"/AbsolutePathWhereScriptIs/yourScript.sql"

STATISTICS MANAGEMENT

Exercise 1

Link to Video on Statistics Management

Perform the following tasks:

1.0. Run the script crea_tablas.sql in order to create tables.
`@"PathTuCarpetaConScripts/crea_tablas.sql"`

1.1. Fill in the tables with the data from the script data2.sql.
Note: The script can be run from the sql worksheet by writing:
@"rutacompleta_fichero.sql"

1.2. Create the following indexes (*):

DX_SEXO ON ACTORS (SEX)

IDX_ANYO ON FILMS (ANYO)

IDX_TITLE ON FILMS (TITLE)

(*) Note that it is much faster and easier to create indexes using SQL statements rather than graphical mode: CREATE INDEX index_name ON table_to_index (column);

1.3. Obtain complete statistics from the ACTORS, FILMS and ACTION tables, including associated indices.

1.4. Analyse the statistical information generated and fill in a table with the following information:

a. Tables: name, number of rows, number of blocks, average length of records.
b. Columns: table and column name, number of different values, minimum and maximum values, number of nulls, average column length, histogram.
c. Indices: name, table, depth, blocks in the sheets, number of different keys, average number of sheet blocks per key, average number of data blocks per key, clustering factor, number of records.

1.5. Can you find something strange in the statistics (strange or inconsistent values in the statistics extracted on the columns)? Hint: look at the attribute statistics, especially the different values and think about the possible values for those attributes.

1.6. Remove the indexes and statistics generated.

1.7. Look at the stored statistical information again and check that the statistics have been deleted and that the indexes created are no longer there (you don't need to provide the results).

QUERY EXECUTION PLANS

When carrying out the exercises below, we will not use **EXPLAIN PLAN** but we will obtain the information, including the execution plan, for the executed queries. To do so, we will activate **autotrace** and **timing** using these two commands:

SET AUTOTRACE ON

SET TIMING ON

Exercise 2

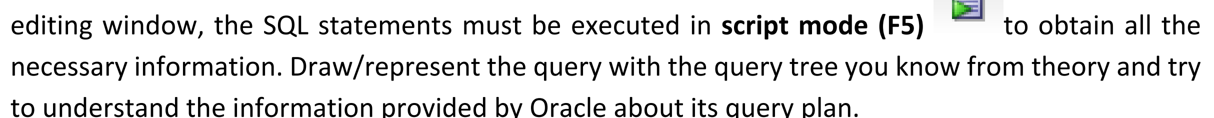
Tables must be filled with the data from the **data2.sql script and without statistics**. Consider the following queries:

```
a. SELECT *
   FROM PELICULAS
   WHERE ANYO > 1992;

b. SELECT *
   FROM PELICULAS P join ACTUACION ACT
   on P.OID = ACT.PELI
   WHERE ANYO > 1992;

c. SELECT P.*
     FROM ACTORES A join ACTUACION X on A.OID = X.ACTOR
     Join PELICULAS P on P.OID = X.PELI WHERE A.NOM = 'Loy, Myrna'
```

Perform the following tasks for each of the above queries (a, b, c):

2.0. Preparation: If you did not finish this in exercise 1.7, delete the statistics now before beginning this exercise (the statistics are calculated automatically at night).

2.1. Obtain your plan and execution time. Note that if **SQL Developer** is used with several queries in the



editing window, the SQL statements must be executed in **script mode (F5)** to obtain all the necessary information. Draw/represent the query with the query tree you know from theory and try to understand the information provided by Oracle about its query plan.

2.2. Is there any contradiction or discrepancy between the information shown in the execution plans and the results obtained? (Hint: in particular, compare the sizes estimated by the system in the execution plan to the real ones). Is there any contradiction between the execution plans and the results obtained?

2.3. Among the information provided by Oracle is a note that says: “Note - dynamic sampling used for this statement”. Investigate and explain the meaning of this message.

2.4. Create statistics for all objects (perform the same procedures as in exercise 1.3 of this laboratory practice).

2.5. Re-obtain the execution plan for each query and compare them to the previous execution plans. What changes can you find and in what cases do they occur? Justify your answers.

Exercise 3

Tables must be filled in with the data from the **data2.sql script** and **without statistics**. Consider the following queries:

```
a. SELECT * FROM PELICULAS WHERE TITULO='8 Seconds';

b. SELECT * FROM PELICULAS WHERE TITULO<='8 Seconds';

c. SELECT * FROM PELICULAS WHERE TITULO>='8 Seconds';

d. SELECT * FROM PELICULAS WHERE TITULO<>'8 Seconds';
```

Perform the following tasks:

3.1 Obtain the execution plan without statistics for each query.
3.2 Generate the corresponding statistics and re-obtain the execution plan for each query Have the execution plans changed? Why?
3.3 Create the index **IDX_TITULO ON PELICULAS (TITLE)** and generate the appropriate statistics.
3.4 Again, obtain the execution plan for all inquiries.
3.5 Indicate in which of these queries the index is now used and explain why you think this happens. Note: check the **clustering_factor** column in the index statistics and compare the value of this statistic of the created index IDX_TITULO with the indexes PK_ACTORES, PK_ACTUACION, _PK_PELICULAS. Find out how this value affects the use of the index.
3.6 Delete the created index.

Exercise 4

Tables must be filled in with the data from the **data2.sql** script and without statistics. Consider the following queries:

```
a. SELECT * FROM PELICULAS WHERE ANYO=1960;
```

```
b. SELECT * FROM PELICULAS WHERE ANYO<1960;

c. SELECT * FROM PELICULAS WHERE ANYO>1960;

d. SELECT * FROM PELICULAS WHERE ANYO<>1960;
```

Perform the following tasks (for queries a, b, c, d):

4.1 Generate the corresponding statistics ensuring that NO HISTOGRAM IS CREATED for the ANYO attribute and obtain the execution plan for each query. Review the plans obtained and write them down. In those plans, look at the number of rows estimated by the system (E-ROWS) and the current number of rows.

```
exec
dbms_stats.gather_schema_stats(ownname=>'ACD0xx',cascade=>true,force=>true,
METHOD_OPT=>'FOR ALL COLUMNS SIZE 1');
```

4.2 Create the IDX_ANYO ON PELICULAS (ANYO) index and generate the appropriate statistics now while ensuring that a histogram is created for the ANYO attribute:

```
exec dbms_stats.gather_schema_stats (ownname => 'ACD0xx', cascade => true,
force => true, METHOD_OPT => 'FOR ALL COLUMNS SIZE 254');
```

Again obtain the execution plan for all the queries and analyse the differences found in the execution plans of the four queries with index. In particular:

- Review and write down the value of E-ROWs and A-ROWS for each query. Are these values very close? Explain why?
- Indicate in which queries (a, b, c, d) the index is used and in which it is not and explain why.
- Review and annotate ***the clustering_factor*** column of the statistics for this new index.

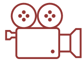
4.3 Run the following query while changing (increasing) the value of the year so that the interval is increasing (steps of 3 in 3 years are suggested):

```
SELECT * FROM PELICULAS WHERE ANYO <1901;
```

Review the execution plans and find when the system stops using the IDX_ANYO index even though it exists. What selectivity factor would be the threshold to turn the index on/off in this query? **Hint:** observe how many rows are retrieved at the time of index deactivation/activation and obtain the proportion of tuples retrieved from the total.

4.4 Delete the created index.

Exercise 5

 Link Video session 2

Preparation (for exercise 5.1): the tables must be empty and loaded with data from the file “data1.sql” script, and the statistics of the tables must be calculated).

Consider the following queries:

```
a. SELECT * FROM ACTORES WHERE SEXO = 'H';

b. SELECT COUNT(*) FROM ACTORES WHERE SEXO = 'H';
```

Perform the following tasks:

5.1 Empty the tables, load them with data from the data1.sql script, and generate the appropriate statistics.

5.2 Create the index of type B-Tree IDX_SEXO ON ACTORES (SEX) and generate the appropriate statistics. **Hint:** A B-tree is the index created by default by Oracle.

5.3 Obtain the execution plan of the queries (a and b) when the IDX_SEXO index has been created. Draw the tree that represents the query, together with the physical plan (mode of access to the table that Oracle will use). Write down the cost of the two queries a) and b) (COST in execution F6). In which query or queries is the index used? Why?

5.4 Delete the index created in step 5.2 and create a new index on the same attribute of the table ACTORES, but a BITMAP index, again generating the appropriate statistics (*).

```
(*)CREATE BITMAP INDEX nombre_indice ON tabla(columna);
```

5.5 Obtain the execution plan of the two queries (a, b) when the BITMAP index has been created. Draw the tree that represents the query, together with the physical plan (mode of access to the table that Oracle will use). Write down the cost of the two queries a) and b) (COST in execution F6). In which query or queries is the index used? Why? Delete the created index.

5.6 Now load data in the file "datos1_null.sql", which contains records with null values for the actors.sex attribute:

```
@".....\crea_tablas_sexonull.sql";
@"....\datos1_null.sql";
--Calcular estadísticas completas
exec dbms_stats.gather_schema_stats(ownname=>'ACD0XX',cascade=>true,force=>true);
```

Let us see how Oracle would solve the following queries on null values, with two indexes of different types (Btree and BITMAP):

```
a. SELECT * FROM ACTORES WHERE SEXO is null;
```

```
b. SELECT COUNT(*) FROM ACTORES WHERE SEXO is null;
```

- Before continuing, do you think either of these two queries could benefit from using an index? Why?
- Create an index type B tree **IDX_SEXO,** recalculate the index statistics, and execute the two queries a) and b). Review the execution plans and indicate whether the created index will be used for either of the two queries. Write down the cost of the two queries (COST parameter when executed with F6). Does it behave as you expected?
- Create a **BITMAP IDX_SEXO_BM** index, recalculate the index statistics, and execute queries a) and b) when this new index is available. Review the execution plans and indicate whether the created index will be used for either of the two queries. Write down the cost of the two queries (COST parameter when executed with F6). Explain this behavior. Review the execution plans and indicates whether the created index will be used for either of the two queries. Does it behave as you would expect?

5.7 Find information about BITMAP indexes and write down the advantages of this kind of index.

5.8 Delete all indexes created.

Exercise 6

Consider the following queries:

```
a. SELECT A.NOM, A.SEXO
   FROM GIISGBD.ACTORES_BASE A
   WHERE EXISTS (
     SELECT *
     FROM GIISGBD.PELICULAS_BASE P join GIISGBD.ACTUACION_BASE
   ACT on P.OID = ACT.PELI
   WHERE P.ANYO = 1980 AND A.OID = ACT.ACTOR);

b. SELECT A.NOM, A.SEXO
   FROM GIISGBD.ACTORES_BASE A
   WHERE A.OID IN (
     SELECT ACT.ACTOR
     FROM GIISGBD.PELICULAS_BASE P join GIISGBD.ACTUACION_BASE
   ACT on P.OID = ACT.PELI
   WHERE P.ANYO = 1980);

c. SELECT A.NOM, A.SEXO
   FROM GIISGBD.ACTORES_BASE A
   WHERE A.OID = ANY (
     SELECT ACT.ACTOR
     FROM GIISGBD.PELICULAS_BASE P join GIISGBD.ACTUACION_BASE
   ACT on P.OID = ACT.PELI
   WHERE P.ANYO = 1980);

d. SELECT DISTINCT A.NOM, A.SEXO
   FROM GIISGBD.PELICULAS_BASE P join GIISGBD.ACTUACION_BASE ACT
   on P.OID = ACT.PELI
        join GIISGBD.ACTORES_BASE A on A.OID = ACT.ACTOR
   WHERE P.ANYO = 1980;
```

Note that the tables used in these queries belong to the GIISGBD schema/user.

The four queries (a, b, c, d) are conceptually identical since they obtain exactly the same results, though they are expressed in different ways: correlated subquery (EXISTS), uncorrelated subquery (with IN = ANY), and join. Perform the following tasks:

6.1 Obtain the execution plan of the four consultations (a, b, c, d). How many tuples are obtained as a result of the above queries? Represent the query tree with the physical plan for each query (a, b, c, d). Write down the cost of execution for each plan. For this, you can see the COST column when the query is run with F6. Are there any differences or similarities between the plans? Why?

6.2 Change the filter of the p.anyo field in all queries to the value 2000. Again obtain the execution plans for the four queries. Is there a difference compared to the previous plans? Why?

Exercise 7

The local user tables must be filled with data from the data1.sql script and the necessary statistics. Consider the following queries:

```
a. SELECT A.NOM, P.TITULO, ACT.PAPEL, P.ANYO
   FROM ACTORES A JOIN ACTUACION ACT ON A.OID = ACT.ACTOR
   JOIN PELICULAS P ON P.OID = ACT.PELI
     WHERE SEXO = ‘H’
     AND P.TITULO LIKE ‘Lost W%’;

b. SELECT A.NOM, P.TITULO, ACT.PAPEL, P.ANYO
   FROM ACTORES_BASE A JOIN ACTUACION_BASE ACT ON A.OID =
   ACT.ACTOR JOIN PELICULAS_BASE P ON P.OID = ACT.PELI
   WHERE SEXO = ‘H’
   AND P.TITULO LIKE ‘Lost W%’;

c. SELECT A.NOM, P.TITULO, ACT.PAPEL, P.ANYO
   FROM ACTORES_BASE A JOIN ACTUACION_BASE ACT ON A.OID =
   ACT.ACTOR JOIN PELICULAS_BASE P ON P.OID = ACT.PELI
   WHERE SEXO = ‘H’
     AND P.TITULO LIKE ‘%Empire%’;
```

Perform the following tasks:

7.1 Obtain the plan and the execution time for queries a, b, c.
7.2 Suggest improvements for executing the queries more efficiently. Test your suggested improvements.

*Note: For testing in query 2, you should copy the tables in the local schemas (create table XXX as select * from giisgbd.XXX) and delete them after the job is finished (drop table XXX).*

VNIVERSITAT DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria ETSE-UV

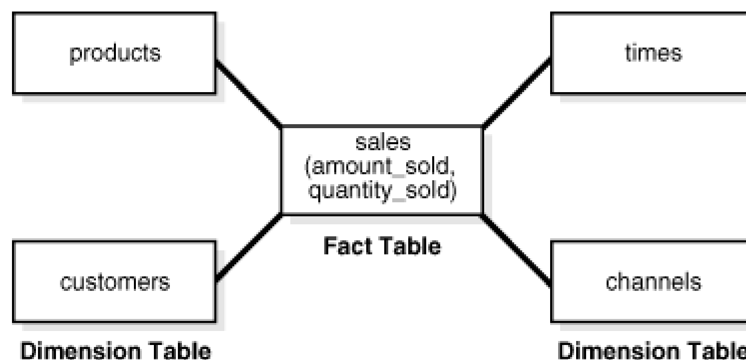
Lab 4: Data Warehousing-
OLAP queries in Oracle

Introduction

In this practice lesson you will become familiar with several tools provided by Oracle for working with data stores, and specifically with SQL extensions, to perform aggregate queries on various dimensions and the creation of special indexes. We will work with a star scheme, which is typical in data stores.

STAR SCHEME

In this laboratory session we will work with a star DB scheme made up of a fact table (SALES) and other tables containing the following dimensions: TIMES, PRODUCTS, CUSTOMERS, AND CHANNELS.



These tables belong to the example SH scheme provided by ORACLE. The tables, which are already created and populated with data, are defined as follows:

SALES(PROD_ID,CUST_ID,TIME_ID,CHANNEL_ID,PROMO_ID,QUANTITY_SOLD,AMOUNT_SOLD)

PRODUCTS(**PROD_ID**,PROD_NAME,PROD_DESC,PROD_SUBCATEGORY,PROD_SUBCATEGORY_ID,PROD_CATEGORY,PROD_CATEGORY_ID, PROD_LIST_PRICE)

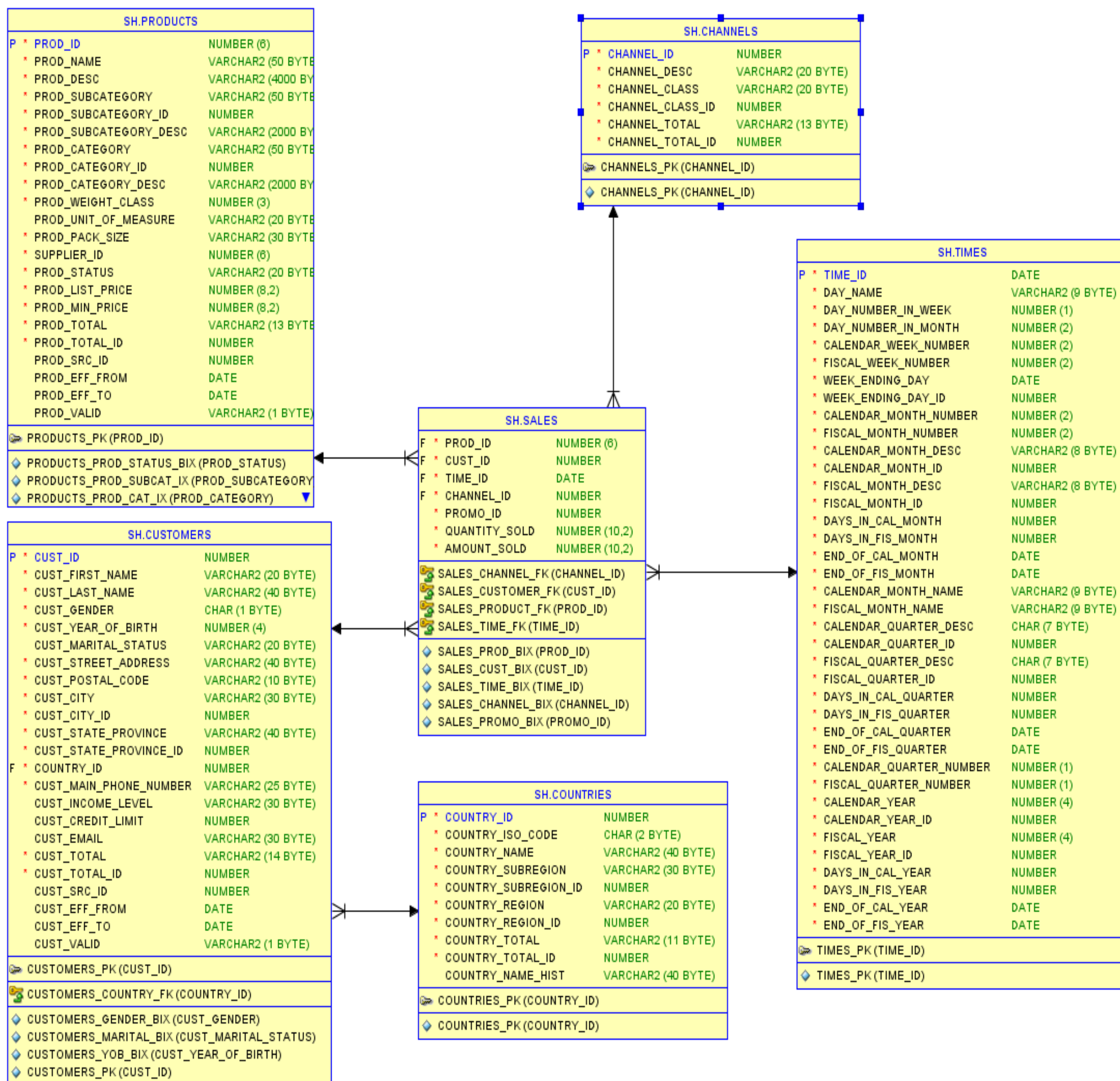
TIMES(**TIME_ID**,DAY_NAME,DAY_NUMBER_IN_WEEK,DAY_NUMBER_IN_MONTH,CALENDAR_WEEK_NUMBER,CALENDAR_MONTH_NAME,CALENDAR_YEAR,DAYS_IN_CAL_YEAR)

CUSTOMERS(**CUST_ID**,CUST_FIRST_NAME, CUST_LAST_NAME, CUST_GENDER, CUST_YEAR_OF_BIRTH, CUST_MARITAL_STATUS, CUST_STREET_ADDRESS,CUST_POSTAL_CODE, CUST_CITY,CUST_CITY_ID,CUST_STATE_PROVINCE,CUST_STATE_PROVINCE_ID, COUNTRY_ID, CUST_MAIN_PHONE_NUMBER,CUST_INCOME_LEVEL,CUST_CREDIT_LIMIT,CUST_EMAIL)

CHANNELS(**CHANNEL_ID**,CHANNEL_DESC, CHANNEL_CLASS, CHANNEL_CLASS_ID,CHANNEL_TOTAL,CHANNEL_TOTAL_ID)

An additional table relates clients to their country of origin to enable inquiries by zones (if this table is added, the scheme becomes a Snowflake scheme).

COUNTRIES(**COUNTRY_ID**,COUTNRY_ISO_CODE,COUNTRY_NAME,COUNTRY_SUBREGION,COUNTRY_SUBREGION_ID,COUNTRY_REGION,COUNTRY_REGION_ID,COUNTRY_TOTAL,COUNTRY_TOTAL_ID)



Part I.

Introduction to SQL for OLAP

The possibility of performing aggregations is a fundamental component of data stores. Oracle provides the following features:

- CUBE and ROLLUP, extensions of the GROUP BY clause.
- 3 GROUPING functions.
- The expression GROUPING SETS.
- Pivoting operations.

To illustrate the use of these operators, we will use the Oracle SH demonstration scheme, which contains tables with useful data for performing aggregation operations.

Example 1:

The company in this example has sales around the world. Imagine we want to extract information: a double entry table showing sales by country and by distribution channel (country_id and channel_desc), while filtering for only two countries (country_iso_code to be 'BR' or 'AR') and two types of distribution channel ('Internet', 'Tele Sales'):

	Country		
Channel	**BR**	**AR**	**Total**
Internet	4,789	2,512	7,301
Tele Sales	5,960	5,235	11,195
Total	10,748	7,747	18,496

In tabular form, the query that would generate this information is

```
SELECT sh.channels.channel_desc, sh.countries.country_iso_code,
       TO_CHAR(SUM(sh.sales.amount_sold), '9,999,999,999') VENTAS$
FROM sh.sales join  sh.customers on
sh.sales.cust_id=customers.cust_id
join  sh.channels on sh.sales.channel_id= sh.channels.channel_id
join SH.COUNTRIES on
sh.customers.country_id=sh.countries.country_id
WHERE  sh.channels.channel_desc IN ('Tele Sales','Internet')
AND sh.countries.country_iso_code IN ('BR','AR')
GROUP BY CUBE(sh.channels.channel_desc,
sh.countries.country_iso_code);
```

Example 2: ROLLUP

ROLLUP enables a SELECT statement to calculate multiple levels of subtotals across a specified group of dimensions. It also calculates general total aggregation.

ROLLUP is an extension of the GROUP BY clause, so its syntax is easy to use. The ROLLUP action is simple: it creates subtotals ranging from the most detailed level to a large total following a grouping list specified in the ROLLUP clause. In this case, the order specified is important.

```
SELECT ch.channel_desc, co.country_iso_code,
TO_CHAR(SUM(sa.amount_sold), '9,999,999,999') VENTAS$
FROM sh.sales sa
  JOIN sh.channels ch on sa.channel_id = ch.channel_id
  JOIN sh.customers cu on sa.cust_id = cu.cust_id
  JOIN sh.countries co on cu.country_id=co.country_id
WHERE  ch.channel_desc IN ('Tele Sales','Internet')
AND co.country_iso_code IN ('BR','AR')
GROUP BY ROLLUP(ch.channel_desc, co.country_iso_code);
```

Example 3: GROUPING SETS

You can selectively specify the set of groups you wish to create by using a GROUPING SETS expression within a GROUP BY clause. This allows precise specification across multiple dimensions without calculating the entire cube. For example:

```
SELECT CHA.channel_desc, TI.calendar_month_desc,
COU.country_iso_code,
      TO_CHAR(SUM(SA.amount_sold), '9,999,999,999') VENTAS
FROM  SH.sales SA, SH.customers CU, SH.times TI, SH.channels CHA,
      SH.countries COU
WHERE SA.time_id=TI.time_id AND SA.cust_id=CU.cust_id AND
      SA.channel_id= CHA.channel_id AND
     CU.COUNTRY_ID = COU.COUNTRY_ID AND
     CHA.channel_desc IN ('Direct Sales', 'Internet') AND
         TI.calendar_month_desc IN ('2000-09', '2000-10') AND
     COU.country_iso_code IN ('GB', 'US')
GROUP BY GROUPING SETS ((CHA.channel_desc, TI.calendar_month_desc,
     COU.country_iso_code), (CHA.channel_desc,
     COU.country_iso_code),
     (TI.calendar_month_desc,   COU.country_iso_code));
```

In this case you specify which groups you want to form. We have selected only two countries, two specific months, and two sales channels.

INTRODUCTION TO SQL ANALITICS AND REPORTING

Oracle presents several analytical calculation functions:

- Rankings and percentiles.
- Mobile windows.
- First and final analysis.
- Statistics with linear regression models.
- OTHERS …

The essential concepts used in analytical functions are:

- **Order of processing.** Query processing through analytical functions is conducted in three stages. First, JOIN, WHERE, GROUP BY and HAVING are performed. Second, the result set is available to the analytical functions. Third, if the query has an ORDER BY clause, the result is sorted and returned in the specified order.
- **Partitions of the result set.** Analytical functions enable users to divide query results into groups of rows called partitions. Partitions are created after the groups defined with the GROUP BY clauses, so they are available for all aggregate results, including sums and averages. Partition divisions can be based on the desired columns or expressions. A set of query results can be divided into a single partition that contains all rows, a few large partitions, or many small partitions each containing few rows.
- **Window definition.** For each row of a partition, a sliding window can be defined. This window determines the range of rows used to perform the calculations for the current row. Window sizes can be based on a physical number of rows or a logical interval such as time. The window has a starting row and an ending row. Depending on its definition, the window may move at one or both ends. For example, a window defined by a cumulative sum function would have its starting row set in the first row of its partition, and its end row would slide from the starting point to the last row of the partition. On the other hand, a window defined for a moving average would change both its starting and ending point but always refer to the current row.
- **Current row.** Each operation performed with the analytical functions is based on a specific row within a partition (current row). The current row serves as a reference for defining the sliding window.

Example 4: RANK AND DENSE_RANK FUNCTIONS

These functions are used to order items in a group: for example, to find the first three products sold in Valencia last year. These two functions have the following syntax:

```
RANK () OVER ([query partition_clause] order_by_clause)

DENSE_RANK OVER ([query partition_clause] order_by_clause)
```

The difference between RANK and DENSE_RANK is that DENSE_RANK does not leave gaps in the ranking, while RANK does. For example, if the result of a competition is ordered by the DENSE_RANK function and three people opt for second place (with the same score), the function would say that the order is [1,2,2,2,3, 4], while RANK would return [1,2,2,2,5,6].

Interesting points in relation to these two functions:

- If a partition is specified, the order is made for each partition.
- If the partition is not specified, the order is calculated on all data.
- The ORDER BY clause specifies the measure on which to obtain the ranking.
- The NULLS FIRST | NULLS LAST clause indicates the position where the null values will be placed.

The following query about the test scheme shows the use of the described functions:

```
SELECT channel_desc, SUM(amount_sold) SALES,
 RANK() OVER (ORDER BY SUM(amount_sold)) AS RANK1,
 RANK() OVER (ORDER BY SUM(amount_sold) DESC NULLS LAST) AS RANK2
FROM SH.SALES, SH.PRODUCTS, SH.CUSTOMERS, SH.TIMES, SH.CHANNELS,
     SH.COUNTRIES
WHERE SH.sales.prod_id=SH.products.prod_id AND
      SH.sales.cust_id=SH.customers.cust_id AND
      SH.customers.country_id = SH.countries.country_id AND
      SH.sales.time_id=times.time_id  AND
      SH.sales.channel_id=channels.channel_id AND
      SH.times.calendar_month_desc IN ('2000-09', '2000-10') AND
      country_iso_code='US'
GROUP BY channel_desc;
```

The following query shows the differences between the two functions explained, is sorted according to the sales made, rounding up to the tens of thousands (-5), to obtain repeated data. The difference is shown as to what the two functions get: RANK () and DENSE_RANK ():

```
SELECT channel_desc, calendar_month_desc, TRUNC(SUM(amount_sold),-5)
      SALES_R, RANK() OVER (ORDER BY TRUNC(SUM(amount_sold),-5) DESC) AS
      RANKING, DENSE_RANK() OVER (ORDER BY TRUNC(SUM(amount_sold),-5)
DESC)
      AS DENSE_RANKING
FROM SH.SALES, SH.PRODUCTS, SH.CUSTOMERS, SH.TIMES, SH.CHANNELS
WHERE SH.sales.prod_id=SH.products.prod_id AND
        SH.sales.cust_id=SH.customers.cust_id AND
        SH.sales.time_id=SH.times.time_id AND
        SH.sales.channel_id=SH.channels.channel_id AND
        SH.times.calendar_month_desc IN ('2000-09', '2000-10') AND
        SH.channels.channel_desc<>'Tele Sales'
GROUP BY channel_desc, calendar_month_desc;
```

The two functions give a position within an order. With Rank (), any repetitions of values are given the same position but the next one is given a later one. With Dense rank (DENSE_RANK ()), this does not occur.

Example 5: CUME_DIST, PERCENT_RANK, NTILE functions.

Other useful functions are the CUME_DIST, the PERCENT_RANK, and N_TILE. The first one gives the position of a specified value within a set of values between [0,1]. It is calculated as the number of values of the set that are before x (including the value itself), divided by the cardinal of the set.

The second function is similar but is calculated using the position of the row within its partition minus 1 divided by the total number of rows in partition -1. This returns a value between [0,1].

The NTILE function calculates quartiles and thirds, etc. This function divides the set into baskets in an orderly fashion and assigns a numerical value to each basket. Each row within the basket is assigned this number.

Below is an example of the latter function in a query that orders the group of tuples returned by a query on the sales made in the year 2000 grouped by calendar month for products in the ‘Electronics’ category. NTILE (4) means that it is divided into four groups.

```
SELECT calendar_month_desc AS MES, SUM(amount_sold)VENTAS,
     NTILE(4) OVER (ORDER BY SUM(amount_sold)) AS CUARTIL
FROM SH.SALES, SH.PRODUCTS, SH.CUSTOMERS, SH.TIMES, SH.CHANNELS
WHERE sales.prod_id=products.prod_id AND
     sales.cust_id=customers.cust_id AND
     sales.time_id=times.time_id AND
     sales.channel_id=channels.channel_id AND
     times.calendar_year=2000 AND prod_category= 'Electronics'
GROUP BY calendar_month_desc;
```

Example 6: Windows

The Window Functions can calculate cumulative aggregate functions, and on sliding windows, while using a row as a reference. They return a value for each row processed (unlike aggregate functions). With these functions you can calculate cumulative versions of the SUM, AVERAGE, COUNT, MAX, MIN functions. They can be used only in the SELECT and ORDER BY clauses. FIRST VALUE AND LAST_VALUE functions are available. These functions allow access to different rows of the same table without needing to make reflexive joins.

```
analytic_function([ arguments ])
   OVER (analytic_clause)

 where analytic_clause =
     [ query_partition_clause ]
     [ order_by_clause [ windowing_clause ] ]

and query_partition_clause =
    PARTITION BY
      { value_expr[, value_expr ]...
      }

 and windowing_clause =
     { ROWS | RANGE }
     { BETWEEN
       { UNBOUNDED PRECEDING
     | CURRENT ROW
     | value_expr { PRECEDING | FOLLOWING }
     }
     AND
     { UNBOUNDED FOLLOWING
     | CURRENT ROW
     | value_expr { PRECEDING | FOLLOWING }
     }
   | { UNBOUNDED PRECEDING
     | CURRENT ROW
     | value_expr PRECEDING
     }
   }
```

For example, the following query performs a cumulative summation of sales for each customer (cust_id) by month (calendar_quarter_desc) and sorts the result by customer. The first aggregate function calculates a

customer's total sales for each month (in the year 2000). The following aggregate function performs the accumulation of those windows within the limits defined by the over() clause.

```
SELECT c.cust_id, t.calendar_quarter_desc, SUM(amount_sold) AS
MonthSales,
     SUM(SUM(amount_sold)
     OVER (PARTITION BY c.cust_id
     ORDER BY c.cust_id,t.calendar_quarter_desc
     ROWS UNBOUNDED PRECEDING) AS CumulativeSales
  FROM SH.SALES S, SH.TIMES T, SH.CUSTOMERS C
  WHERE s.time_id=t.time_id AND s.cust_id=c.cust_id AND
        t.calendar_year=2000
  GROUP BY c.cust_id, t.calendar_quarter_desc
  ORDER BY c.cust_id, t.calendar_quarter_desc;
```

The first rows obtained by the above query are shown in the table below, where 'current row' is the record at which we are located and for which the window is defined. In the query, the window includes the current record we are calculating and all previous rows within the same partition.

cust_id	calendar_quarter_desc	MonthSales	CumulativeSales
4	2000-02	720,11	720,11
4	2000-03	1680,1	2400,21
4	2000-04	240,61	2640,82
6	2000-02	3445,17	3445,17
6	2000-03	6587,4	10032,57
7	2000-01	64,41	64,41
7	2000-02	53,09	117,5
9	2000-04	10189,62	10189,62
13	2000-02	990,19	990,19
13	2000-03	7477,01	8467,2
13	2000-04	1934,04	10401,24

The following example shows the use of windows when the window limits are defined logically. It calculates, for each customer and month, the cumulative sum during that month. We also want to calculate, for each row of the result, the sum of the accumulated sales in the previous, current, and next months. To do so, we use the logical definition of the window (RANGE) and indicate how many months before and after we wish to consider for the calculation. Note: It is extremely important, if the window definition is logical, that the attribute used to order rows is a number or a date (DATE).

```
SELECT sa.cust_id, t.calendar_quarter_desc,t.calendar_quarter_id,
SUM(amount_sold) AS MonthSales,
SUM(SUM(amount_sold))
OVER (PARTITION BY sa.cust_id
ORDER BY t.calendar_quarter_id
range between 1 PRECEDING and 1 FOLLOWING) AS CumSalesPrePos
  FROM SH.SALES SA JOIN SH.TIMES T ON sA.time_id=t.time_id
  WHERE t.calendar_year=2000
  GROUP BY sa.cust_id, t.calendar_quarter_desc,
calendar_quarter_id
     ORDER BY sa.cust_id, t.calendar_quarter_desc,
calendar_quarter_id;
```

cust_id	calendar_quarter_desc	calendar_quarter_id	MonthSales	CumuSalesPrePost
33	2000-01	1777	60,84	102,62
33	2000-02	1778	41,78	102,62
34	2000-01	1777	2063	2063
34	2000-04	1780	395,61	395,61
37	2000-02	1778	664,52	3545,99
37	2000-03	1779	2881,47	4727,17
37	2000-04	1780	1181,18	4062,65
41	2000-03	1779	3121,84	3140,5
41	2000-04	1780	18,66	3140,5
42	2000-03	1779	919,85	6493,22
42	2000-04	1780	5573,37	6493,22
44	2000-01	1777	2717,77	2717,77
44	2000-04	1780	345,76	345,76

current row
1 preceding
1 following

Table 2: Several rows resulting from the above SQL sentence execution.

INTRODUCTION TO MATERIALIZED VIEWS IN ORACLE

Materialized views are used to improve performance. However, the overhead associated with view materialization can become a major system-management problem. Some tasks relating to view management are:

- Identifying which materialized views to create initially.
- Indexing materialized views.
- Ensuring that all materialized views and associated indexes are correctly updated whenever the database is updated.
- Checking which materialized views have been used.
- Determining how effectively each materialized view improves performance.
- Measuring the space being used by materialized views.
- Determining whether more materialized views should be created.
- Determining whether any materialized views already created should be deleted.

In Oracle there are three basic types of materialized views:

- Materialized views with aggregates.
- Materialized views with only JOIN operations.
- Nested materialized views.

Example 7: Creation of materialized views.

To create materialized views, we use the statement:

```
CREATE MATERIALIZED VIEW
```

For example, the following statement creates a materialized view that calculates the total money spent from the sales table for each product:

```
CREATE MATERIALIZED VIEW product_sales_mv
REFRESH COMPLETE ON DEMAND
AS
SELECT p.prod_name, SUM(s.amount_sold) AS dollar_sales
FROM sales s, products p WHERE s.prod_id = p.prod_id
GROUP BY p.prod_name;
```

The previous query updates the view completely when explicitly requested. Other valid options for creating the view are:

```
REFRESH COMPLETE ON COMMIT
REFRESH FAST ON COMMIT
REFRESH COMPLETE ON DEMAND
REFRESH FAST ON DEMAND
REFRESH FORCE ON DEMAND
```

Another interesting option in relation to views is to enable queries to be rewritten: ENABLE QUERY REWRITE. This option aim to determine whether the response will be expedited by using the materialized view, when the original query does not use the view directly. If so, the query is rewritten using the materialized view.

The following system privileges are required to create materialized views of source tables located in other schemas:

```
CREATE MATERIALIZED VIEW system privilege
CREATE ANY MATERIALIZAED VIEW
GLOBAL QUERY REWRITE system privilege
```

The following table shows the restrictions on using the added functions when the REFRESH FAST option is used. In this table, X is the aggregate function you wish to use, Y is then the mandatory function to also use, while Z is optional but recommended.

X	Y	Z
COUNT(expr)	-	-
MIN(expr)	-	-
MAX(expr)	-	-
SUM(expr)	COUNT(expr)	-
SUM(col), col has NOT NULL constraint	-	-
AVG(expr)	COUNT(expr)	SUM(expr)
STDDEV(expr)	COUNT(expr) SUM(expr)	SUM(expr * expr)
VARIANCE(expr)	COUNT(expr) SUM(expr)	SUM(expr * expr)

REFERENCES:

https://docs.oracle.com/database/121/DWHSG/concept.htm#DWHSG-GUID-452FBA23-6976-4590-AA41-1369647AD14D

https://docs.oracle.com/database/121/DWHSG/part3.htm#DWHSG8110

https://docs.oracle.com/database/121/DWHSG/part5.htm#DWHSG8493

Part II. Development

This practice lesson will be developed using the Oracle 10g database installed on the **pokemon.uv.es** server, for which all students have the necessary access credentials.

INTRODUCTION TO OLAP OPERATORS IN ORACLE

Exercise 1: GROUP BY CUBE (related to Example 1)

1. What is the number of rows in the tables of the SH schema shown in the schema on page 2 of this document? Make the necessary queries to find or access the statistics of these tables. Make the necessary queries to find them.
2. What basic **group by** type queries would you need to execute to obtain the result in Example 1 (you only need to write the different **group by** clauses).
3. Use the CUBE operator to generate a double entry table (in table format) that shows the sum of sales (SH.SALES.AMOUNT_SOLD) by channel type (SH.CHANNELS.CHANNEL_DESC) and product category (attribute PRODUCTS.PROD_CATEGORY). **Note:** Information of this type is obtained with the CUBE operator. Note how many rows the query returned. How many rows would you expect for this query? Does this value match? Why?
4. Reviewing the result of the query in exercise 1.3, how much money is invested in total (sum of SALES.AMOUNT_SOLD) in products sold by the **'Partners'** channel and how much in products sold by **'Tele Sales'**? Which product category has been sold the most by 'Direct Sales'?

Exercise 2: ROLLUP (related to Example 2)

Execute the query from Example 2 above and interpret the null values that appear in some of the columns.

1. Use the ROLLUP operator to obtain sales (sum of amount_sold in sales) by customers, grouping within the hierarchy of customer addresses (country_region, country_subregion, country). If you look at the results, say that regions are formed only by a sub-region, so there are several rows in which the subtotals coincide.
2. We now want to see sales information in more detail, i.e. by region, sub-region, country, province, and city. What operation are we performing if we start from the query in exercise 2.1? Perform the query to obtain the requested values.
3. Again, using the ROLLUP operator, obtain for the registered Spanish cities and provinces (COUNTRY_ISO_CODE = 'ES') the sum of sales per Spanish province and city. Review the result of the query and find in which Spanish province the most sales have been made.
   Note: attributes CUST_STATE_PROVINCE, and CUST_CITY correspond to the province and city, respectively, of the customer.

Exercise 3: GROUPING SETS (related to Example 3)

1. Make a query that calculates total sales (in the months of September and October 2000), grouping by the following characteristics only:
   ✓ month of sale and region of sale

- ✓ month of sale and distribution channel
- ✓ subcategory of product

Note: Possible useful attributes for setting the groups are: `calendar_month_desc,country_region,channel_desc,prod_subcategory.`

From the outcome of this query, answer the following questions:

2. In Europe, how many sales were made in August 2000, and how many sales were made via the 'Direct Sales' channel in the same month?
3. What cardinality (number of tuples) does the query result have?
4. To estimate this cardinality, what additional information do you need? Also, how would you calculate the number of potential tuples (the maximum number of records the result can contain)? For this query, what is the maximum number of tuples that can be retrieved?

INTRODUCTION TO SQL FOR ANALYTICS AND REPORTING

Exercise 4: RANK and DENSE_RANK functions (related to Example 4)

1. As the RANK() function can sort by partitions, use the corresponding clause to make a query that sorts, within each distribution channel, the total sales (sum(amount_sold)) in the years in which there are records. This query attempts to determine the year in which the most sales have been made within a given sales channel.

   Note: The attribute CALENDAR_YEAR from table sh.times contains the year with four digits and can be used. The attribute which contains distribution channel information is SH.CHANNELS.CHANNEL_DESC

   The optional [QUERY PARTITION_CLAUSE] clause has the form:

   `PARTITION BY column | expression`

2. From the result of the execution of the previous query, indicate the year in which the most sales were made for each existing distribution channel. Modify the previous query to order the years by the number of total windows (from highest to lowest) regardless of the distribution channel. Which year appears in the first position now?
3. In the query in exercise 4.1, add a column that calculates the DENSE_RANK() function on a window where the order is established once the total sales are rounded up to million units and compare this order with that obtained with the RANK() function. What differences do you notice?

   NOTE: To round up to the nearest million use the ROUND(number,-6) function.

   Example: *`ROUND(15569726,21,-6) -> 16000000`*

Exercise 5: PERCENT_RANK function (related to Example 5)

1. Write a query that calculates the percentile (PERCENT_RANK) of the total sales by customers grouped by country. From this result, which countries are above the 85th percentile (value 0.85 of the PERCENT_RANK function)? Explain in your own words what it means to be above this percentile.

Exercise 6: Windows (related to Example 6)

1. Write a query that computes the purchases made by the customer with customer code 2 (CUST_ID = 2) for all months in 1998 (CALENDAR_YEAR = 1998). Note that two attributes in the SH.TIMES table store information about the month: calendar_month_desc and calendar_month_id.
2. Once you have this query, use a window to calculate the average purchases using the previous query (6.1) as a base. The window for computing the average is defined on the sales in the month we are currently in along with the previous two months. **Note:** Think about whether the window definition for this case should be done using the physical window definition or the logical one (the ROWS clause or the RANGE clause).
3. Perform a query that calculates the total sales made within the sub-region COUNTRY_SUBREGION= 'Western Europe' grouped by country, province and city.
4. Modify the query in exercise 6.3 to obtain and display, for each row of the previous set, two additional columns, one with the maximum for each province and one with the total sales for each country.

INTRODUCTION TO MATERIALIZED VIEWS IN ORACLE

Exercise 7: Materialized views (related to Example 7)

1. Execute the following query, which calculates the total amount spent (quantity_sold*amount_sold) for each product and quantity of products. The query displays the product name, amount spent, and quantity of product purchased (for all countries, customers, and time). Note the time it takes to resolve the query and the cost of execution (Auto-tracking F6 function).

```
select PRO.PROD_NAME, SUM(SA.QUANTITY_SOLD*SA.AMOUNT_SOLD) as
moneytotal, SUM(SA.QUANTITY_SOLD) as totalAmount
from SH.SALES SA JOIN SH.PRODUCTS PRO on
SA.PROD_ID=PRO.PROD_IDGROUP BY PRO.PROD_NAME;
```

2. Now create a materialized view (name: PRODUCT_COUNTRY_CHANNEL_MV) that, for each COUNTRY_ID, product name and distribution channel type, stores the amount of money invested and the number of products sold. Create the view so that it is fully refreshed when needed (REFRESH COMPLETE ON DEMAND).

3. Review the view and use it to solve exercise 7.1. Write down the cost of executing the query when the materialized view is used (COST parameter in F6, automatic tracing).

4. Make the following query and in the execution plan (F6) check which table/view the systems needs to access to solve it and find the cost of execution.

```
SELECT t.calendar_month_desc, SUM(s.amount_sold)
FROM sh.sales s, sh.times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

5. Now create the following view with the QUERY REWRITE option:

```
CREATE MATERIALIZED VIEW ventas_por_mes
ENABLE QUERY REWRITE AS
SELECT t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM sh.sales s, sh.times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

6. Now run the query from exercise 7.4 again and review the execution plan the system uses to solve the query (F6), and its associated cost. What do you observe?

VNIVERSITAT DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria ETSE-UV

Lab 5: Cassandra

The tasks to be performed relate to basic work with a column-oriented NoSQL database:

• Create and operate with a column-oriented database.

• Practise select queries and insert, update and delete operations.

About Cassandra

Apache Cassandra, a top level Apache project born at Facebook and built on Amazon's Dynamo and Google's BigTable, is a distributed database for managing large amounts of structured data across many commodity servers, while providing highly available service and no single point of failure. Cassandra offers capabilities that relational databases and other NoSQL databases simply cannot match such as: continuous availability, linear scale performance, operational simplicity and easy data distribution across multiple data centers and cloud availability zones.

Cassandra's architecture is responsible for its ability to scale, perform, and offer continuous uptime. Rather than using a legacy master-slave or a manual and difficult-to-maintain sharded architecture, Cassandra has a masterless "ring" design that is elegant, easy to setup, and easy to maintain.

In Cassandra, all nodes play an identical role; there is no concept of a master node, with all nodes communicating with each other equally. Cassandra's built-for-scale architecture means that it is capable of handling large amounts of data and thousands of concurrent users or operations per second -even across multiple data centers- as easily as it can manage much smaller amounts of data and user traffic. Cassandra's architecture also means that, unlike other master-slave or sharded systems, it has no single point of failure and therefore is capable of offering true continuous availability and uptime — simply add new nodes to an existing cluster without having to take it down.

Many companies have successfully deployed and benefited from Apache Cassandra including some large companies such as: Apple, Comcast, Instagram, Spotify, eBay, Rackspace, Netflix, and many more. The larger production environments have PB's of data in clusters of over 75,000 nodes. Cassandra is available under the Apache 2.0 license.

REMARKABLE DETAILS

Cassandra is considered a schema-less data-store, but it is necessary to perform some configuration specific to your application.

Keyspaces are the upper-most namespace in Cassandra and typically you'll see exactly one for each application. For each keyspace there are one or more column families (like tables). A column family is the namespace used to associate records of a similar kind. Cassandra gives you record-level atomicity within a column family when making writes, and queries against them are efficient. These qualities are important to keep in mind when designing your data model, as you'll see in the discussion that follows.

In Cassandra there is no referential integrity, and the lack of support for secondary indexing makes it difficult to efficiently perform joins, so you must denormalize. You're forced to think in terms of the queries you'll perform and the results you expect since this is likely what the model will look like.

Another important distinction from traditional databases in that the order records are sorted is a design decision, and not something that can easily be changed later.

A relational database will scan your table sequentially when performing a SELECT-WHERE like this, and since records are distributed throughout a Cassandra cluster based on key, the equivalent could mean contacting more than one node (possibly many). However, even with all of the data on a single machine, there comes a point when such an operation will become inefficient with a relational database, making it necessary to index the username attribute. Cassandra doesn't currently support secondary indices like this.

First steps

Install Cassandra in your system

in Windows / in Linux (see instructions).

Start the Cassandra service or verify that the Cassandra service is running

in Windows / in Linux (sudo service cassandra { start | stop | status).

Open the Cassandra CQL shell (CQL Shell / cqlsh). This should provide a prompt cqlsh>

Execute the following commands and observe and interpret their results:

1. help
2. help describe
3. describe cluster
4. show host
5. describe keyspaces
6. use system_schema;
7. describe tables
8. describe table tables
9. select keyspace_name, table_name from tables;
10. select keyspace_name, table_name from tables where keyspace_name='system_auth';
11. select keyspace_name, table_name from tables where table_name='roles';
12. select keyspace_name, table_name from tables where table_name='roles' ALLOW FILTERING;

Exit the CQL command interface (with exit).

Exercises

This practice session will work with the data from a fictitious KILLRVIDEO application that enables users to include information about the videos uploaded to a video-distribution system.

Deliver a text file as with your solution to the practice. Transcribe the order used in each step and copy the result or describe it if it is very long.

First save the working files to your disk. The following exercises will be performed using the Cassandra CQL Shell text interface. In each case, transcribe the order used and copy or describe the result.

EXERCISE 1. CREATE A SPACE AND A TABLE

Objectives:

- Create a keyspace for KillrVideo.
- Create a table to store video metadata.
- Load the video table with data from a CSV file.

To perform these tasks, we will start by creating a table schema and loading certain data from the videos.

The video metadata consist of:

Column Name	Data Type
video_id	timeuuid
added_date	timestamp
description	text
title	text
user_id	uuid

Steps

1.1) In CQLSH, create a keyspace (with CREATE KEYSPACE) named killrvideo and use USE to change to that keyspace. Use SimpleStrategy as the replication class and a replication factor of 1.

1.2) Create (CREATE TABLE) the videos table with the required structure, with video_id as the primary key.

1.3) Load the newly created table with the videos.csv file using the COPY command.

```
COPY videos FROM 'videos.csv' WITH HEADER=true;
```

COPY does not require column names when the schema columns of the target table match the order of the fields in the CSV source file.

1.4) Use SELECT to verify that the data have been loaded properly. Include LIMIT to retrieve only the first 10 rows.

1.5) Use SELECT COUNT (*) to count the number of rows in the table. The number must match the number of rows indicated by the COPY command.

1.6) Use SELECT with WHERE to get the data of a video with the title = 'Become An Internet Enterprise'. What should you do to get the result and why?

EXERCISE 2. TABLES WITH COMPOUND KEY

Objectives:

- Create a new table that enables you to find the videos by title and year (bear in mind that there may be videos with the same title but from different years)

We need to consult the data of the videos by searching by title and year (do not use ALLOW FILTERING if it is unnecessary to do so). The columns for the new table are:

Column Name	Data Type
title	text
added_year	int
added_date	timestamp
Description	text
user_id	uuid
video_id	uuid

Steps

2.1) Create a second table in the killrvideo keyspace called videos_by_title_year with the above-mentioned structure, setting the combination of title and added_year as primary key (since two videos with the same title are not allowed in the same year). Write for this: PRIMARY KEY (title, added_year).

2.2) Load the data from the videos_by_title_year.csv file using the COPY command.

2.3) Get the data of videos with a specific title and year, such as:

What is DataStax Enterprise?   2013

2.4) Get the data of videos with the title 'What is DataStax Enterprise?'

2.5) Check the data of videos added in 2013.

2.6) Drop the table (DROP TABLE), create it again but now writing PRIMARY KEY ((title, added_year)) – pay attention to the double parentheses! – and execute the previous steps from 2.2. to 2.5. Which answers change compared to the previous ones, and why?

Exercise 3. Clustering columns

Objectives:

- Create a table that enables you to search for videos by tag and range of years.

You want to assign tags to the videos and then search for videos with a specific tag and for those with a specific tag and from a specific year or range of years (before A, or after B, or between C and D, etc.). To do so, we will create a new table.

Steps

3.1) Create the videos_by_tag_year table with the following command

```
CREATE TABLE videos_by_tag_year (
  tag text,
  added_year int,
  video_id timeuuid,
  added_date timestamp,
  description text,
  title text,
  user_id uuid,
  PRIMARY KEY ((tag), added_year, video_id)
) WITH CLUSTERING ORDER BY (added_year desc, video_id asc);
```

Why is the primary key defined this way?

3.2) Load the data from the videos_by_tag_year.csv file.

3.3) Obtain the number of rows in the videos_by_tag_year table.

3.4) Obtain the data of videos with tag = 'nosql' from the year 2012.

3.5) Obtain the data of videos with tag = 'nosql' and year before 2015.

3.6) Obtain the data of videos that have the tag 'nosql'.

3.7) Obtain the data of videos added between 2010 and 2014, both years inclusive.

3.8) Explain which queries can be carried out efficiently (without the need for ALLOW FILTERING), and why.

3.9) Find out how many videos there are, separated by years, with the tag 'nosql'.

Exercise 4. Adding columns and defining new types

Objectives:

- Alter an existing table and add additional columns.
- Create new user-defined types.
- Load additional records or fields into a table that already contains data.

After reviewing the structure of the tables, we want to add the tags for a video to the original table videos and include information about the video encoding. To do so, we will add a column of type SET and another with a compound type defined by the user.

> *Collection columns are multi-valued columns designed to store a small amount of data that are always retrieved in their entirety. A collection cannot be nested inside another collection.*
>
> > *SET: Typed collection of unique values / Ordered by values.*
> >
> > *LIST: Typed collection of non-unique values / Ordered by position.*
> >
> > *MAP: Typed collection of key-value pairs / Ordered by unique keys.*

Steps

4.1) Execute the TRUNCATE command to delete the data from the video table.

4.2) Alter the videos table by adding a set column field, with the command:

```
ALTER TABLE videos ADD tags SET <TEXT> ;
```

4.3) Load the data from the videos_with_tags.csv file using the COPY command.

4.4) Get the title and tags of the first five videos.

4.5) Create a new data type to save the encoding of the videos, with the command

```
CREATE TYPE video_encoding (
  encoding TEXT,
  height INT,
  width INT,
  bit_rates SET <TEXT>
);
```

4.6) Alter the videos table by adding a column with name encoding and type FROZEN <video_encoding>.

4.7) Without deleting the existing contents, load the encoding data of the videos contained in the videos_encoding.csv file, using COPY as: COPY videos (video_id, encoding) FROM…

4.8) Obtain the date, tags and encoding of videos with the title 'Getting Started with Spark & Cassandra'

Exercise 5. Counters in Cassandra

Objectives:

- Create a new table that uses the counter type.
- Load data into the table with counters with a CQL file.
- Perform queries to verify the functionality of a counter.

You want to keep a consistent count of the number of videos that contain a tag in a particular year. Whenever a tag is added to a video, a transaction should be performed that reads the number of videos with that tag for that year, increases it by one, and updates the database. However Cassandra, being a distributed database that allows simultaneous operations and does not manage transactions, does not normally ensure total consistency. The counter type is the way to solve this problem.

Steps

5.0) Consult and explain what the Cassandra counter type is and what it is used for.

5.1) Create a new table videos_count_by_tag with (TEXT) and added_year (INT) columns, which form the primary key, and with another column named video_count of type counter.

5.2) Execute with the SOURCE command the SQL commands contained in the videos_count_by_tag.cql file.

5.3) Obtain the number of videos tagged 'nosql' in each year after 2011.

If a new video is inserted in the videos_by_tag_year table, the video count must also be updated by tag in the videos_count_by_tag table:

5.4) Insert in the videos_by_tag_year table the data of a clip with the label 'nosql', year of addition 2015, identifier of the video with the timeuuid of now() and title 'nosql for all' (the other fields remain null)

5.5) Execute the update that must be done in the videos_count_by_tag table (with UPDATE) after including (in 5.4) a new video with the tag 'nosql' from the year 2015.

5.6) Now check the new number of videos with the tag 'nosql' from the year 2015.

Exercise 6. Query-driven design (and denormalization)

Objectives:

- Create denormalized tables that allow / optimize the queries required by our system.

According to the relational model, if we have videos and actors, we will have one table of videos, one of actors and one to reflect the N:N relationship between them, with the primary keys of each entity. Using SQL through JOINs will enable us to consult the videos in which a certain actor has participated.

However, Cassandra does not allow JOINs, so we must denormalize the tables based on the queries we wish to allow.

In this case, the query is made to obtain a list of videos in which a certain actor participates, ordered by the date of addition (the most recent first) and bearing in mind that an actor can play one or more characters ('character') in the same video.

Steps

6.1) Create a videos_by_actor table with the corresponding CREATE TABLE command to allow the above-mentioned query (we will load the data from the videos_by_actor.csv file).

The required fields and types are (in alphabetical order, which does not necessarily have to be the order in the table layout):

actor text,

added_date timestamp,

character_name text,

description text,

encoding frozen <video_encoding>,

tags set <text>,

title text,

user_id uuid,

video_id timeuuid,

You must define the PRIMARY KEY and CLUSTERING ORDER that are appropriate and necessary.

6.2) Load the videos_by_actor table with the data from the videos_by_actor.csv file.

(This will take time as there are over 80,000 rows!) How many rows have been inserted?

6.3) Obtain the titles in which Jerry Lewis performs as an actor and the characters he plays.

6.4) Obtain the first 12 titles that appear in which Miranda Richardson performs, with the date and name of the character she played. In what order do they appear?

Lab 6: MongoDB

The tasks to be performed relate to basic work with a document-oriented NoSQL database:

- Operate with a database of documents, where a document is understood to be semi-structured information of key-value pairs.
- Create a database on a schema-less system.
- Insert, update and delete operations.
- Conduct basic and advanced select query operations.

The file municipios.json contains data on Spanish municipalities, including autonomous community, province, name of municipality, geographical coordinates (latitude and longitude), altitude, number of male, female and total inhabitants, and post codes. The file format is JSON and the file can be imported directly into a MongoDB database.

Introduction

MongoDB is a document database that provides high performance, high availability, and easy scalability.

- *Document Database*
  - *Documents (objects) map nicely to programming language data types.*
  - *Embedded documents and arrays reduce need for joins.*
  - *Dynamic schema makes polymorphism easier.*
- *High Performance*
  - *Embedding makes reads and writes fast.*
  - *Indexes can include keys from embedded documents and arrays.*
  - *Optional streaming writes (no acknowledgments).*
- *High Availability*
  - *Replicated servers with automatic master failover.*
- *Easy Scalability*
  - *Automatic sharding distributes collection data across machines.*
  - *Eventually-consistent reads can be distributed over replicated servers.*
- *Advanced Operations*
  - *With MongoDB Management Service (MMS) MongoDB supports a complete backup solution and full deployment monitoring.*

MongoDB Data Model

A MongoDB deployment hosts a number of databases. A database holds a set of collections. A collection holds a set of documents. A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

MongoDB Queries

Queries in MongoDB provide a set of operators to define how the find() method selects documents from a collection based on a query specification document that uses a combination of exact equality matches and conditionals using a query operator.

Deployment Architectures

Although MongoDB supports a "standalone" or single-instance operation, production MongoDB deployments are distributed by default. Replica sets provide high performance replication with automated failover, while sharded clusters make it possible to partition large data sets over many machines transparently to the users.

MongoDB users combine replica sets and sharded clusters to provide high levels of redundancy for large data sets transparently for applications.

MongoDB Design Philosophy

MongoDB wasn't designed in a lab. We built MongoDB from our own experiences building large scale, high availability, robust systems. We didn't start from scratch, we really tried to figure out what was broken, and tackle that. So the way I think about MongoDB is that if you take MySql, and change the data model from relational to document based, you get a lot of great features: embedded docs for speed, manageability, agile development with schema-less databases, easier horizontal scalability because joins aren't as important. There are lots of things that work great in relational databases: indexes, dynamic queries and updates to name a few, and we haven't changed much there. For example, the way you design your indexes in MongoDB should be exactly the way you do it in MySql or Oracle, you just have the option of indexing an embedded field.

—Eliot Horowitz, MongoDB CTO and Co-founder

- *New database technologies are needed to facilitate horizontal scaling of the data layer, easier development, and the ability to store order(s) of magnitude more data than was used in the past.*
- *A non-relational approach is the best path to database solutions which scale horizontally to many machines.*
- *It is unacceptable if these new technologies make writing applications harder. Writing code should be faster, easier, and more agile.*
- *The document data model (JSON/BSON) is easy to code to, easy to manage(dynamic schema), and yields excellent performance by grouping relevant data together internally.*

- *It is important to keep deep functionality to keep programming fast and simple. While some things must be left out, keep as much as possible – for example secondaries indexes, unique key constraints, atomic operations, and multi-document updates.*
- *Database technology should run anywhere, being available both for running on your own servers or VMs, and also as a cloud pay-for-what-you-use service.*

Key MongoDB Features

• Flexibility

MongoDB stores data in JSON documents (which we serialize to BSON). JSON provides a rich data model that seamlessly maps to native programming language types, and the dynamic schema makes it easier to evolve your data model than with a system with enforced schemas such as a RDBMS.

• Power

MongoDB provides a lot of the features of a traditional RDBMS such as secondary indexes, dynamic queries, sorting, rich updates, upserts (update if document exists, insert if it doesn't), and easy aggregation. This gives you the breadth of functionality that you are used to from an RDBMS, with the flexibility and scaling capability that the non-relational model allows.

• Speed/Scaling

By keeping related data together in documents, queries can be much faster than in a relational database where related data is separated into multiple tables and then needs to be joined later. MongoDB also makes it easy to scale out your database. Autosharding allows you to scale your cluster linearly by adding more machines. It is possible to increase capacity without any downtime, which is very important on the web when load can increase suddenly and bringing down the website for extended maintenance can cost your business large amounts of revenue.

• Ease of use

MongoDB works hard to be very easy to install, configure, maintain, and use. To this end, MongoDB provides few configuration options, and instead tries to automatically do the "right thing" whenever possible. This means that MongoDB works right out of the box, and you can dive right into developing your application, instead of spending a lot of time fine-tuning obscure database configurations.

Exercises

Previous (mongoimport)

From the operating system command line (not from the mongo shell), run the mongoimport program to import the data into MongoDB from the municipios.json file.

Use the appropriate mongoimport syntax to store the data from the municipios.json file in the mydb database in the 'municipios' collection.

Exercise 0 (analysis of examples)

Enter the mongo online interface, run the samples listed in the samples file, explain what query they perform, and write the equivalent SQL query that would be done on a relational database.

a)

```
db.municipios.find({poblacion:'Valencia'})
```

b)

```
db.municipios.find({poblacion:'Valencia'}, {poblacion:1,
provincia:1, comunidad:1, _id:0})
```

c)

```
db.municipios.find({comunidad:'Asturias'}).limit(5)
```

d)

```
db.municipios.find({comunidad:'Asturias'}).count()
```

e)

```
db.municipios.distinct('comunidad')
```

f)

```
db.municipios.find( {habitantes:{$gte:500000}} , {poblacion:1,
habitantes:1, _id:0} ).sort({habitantes:-1})
```

g)

```
db.municipios.find( { zip : {$exists : false }} , {poblacion:1 ,
_id:0} )
```

h)

```
db.municipios.find({} , {poblacion:1, localizacion:1,
_id:0}).sort({ "localizacion.0" : -1}).limit(1)
```

i)

```
db.municipios.aggregate( { $group : { _id : "", totalVarones :
{$sum: "$varones"}, totalMujeres : {$sum: "$mujeres"} } })
```

j)

```
db.municipios.aggregate( { $group : { _id : "$provincia",
totalPob : {$sum: "$habitantes"} } } , { $sort : {totalPob : -1}
} )
```

k)

```
db.municipios.aggregate( { $match : { altitud : { $gte : 1000
}}} , { $group : { _id : "$provincia", numPob : { $sum : 1 }}},
{ $match : { numPob : { $gte : 20 }}}, {$sort : {numPob:-1}})
```

Exercise 1

Obtain the data are for a specific population (e.g. Jalance).

Exercise 2 (count)

Check the number of documents in the 'municipios' collection.

Exercise 3 (insert)

Insert the following dummy population into the database and check how it was inserted:

population: 'Batman', inhabitants: 3, males: 3, females: 0

Exercise 4 (update and $ set)

A woman has registered in the municipality. Update the data with one more inhabitant and one woman, and check that it has been updated.

Exercise 5 (update)

Not all documents contain the zip field. Include the post code for Zahara (Cádiz), which is 11688, and check that it has been updated properly.

Exercise 6 (remove)

Delete the document from the population inserted in exercise 3 and check that it has been deleted.

Exercise 7 (find and limit)

Get a list of any three municipalities in the province of Alacant.

Exercise 8 ($ gt)

Obtain data for municipalities with over 1,000,000 inhabitants.

Exercise 9 (projection)

Show ONLY the names of any four municipalities.

Exercise 10 ($lte selection and projection)

Show the names of the municipalities with their province and number of inhabitants (no other fields) in municipalities with at most 10 inhabitants.

Exercise 11 (distinct)

Obtain the names of all provinces without duplicates.

Exercise 12 (count)

Obtain the total number of municipalities in the autonomous community of Extremadura.

Exercise 13 (aggregate, $ match, $ group, $ sum)

Obtain, in the same query, the total number of inhabitants, the total number of men, and the total number of women in the Valencian autonomous community ('Valencia').

Exercise 14 (find, sort, limit)

Obtain the data for the two highest municipalities in Spain.

Exercise 15 ($ exists)

Show only the name of the municipality and the province of five municipalities without a post code (zip) in the document.

Exercise 16

Obtain the name of the municipality, province, autonomous community, and location of:

a) the municipality located in the northernmost part of Spain.

b) the municipality located in the westernmost part of Spain.

c) the municipality located furthest south of the Valencian Community ('Valencia').

Exercise 17

Obtain a list of the total populations of the Spanish autonomous communities. Order this list from the highest total population to the lowest.

Exercise 18

Calculate the total population, the total number of males, and the total number of females for Spain.

Exercise 19 (aggregate, $ group, $ avg)

Obtain the average population of the municipalities in each province and order the provinces in increasing order of population.

Exercise 20

Obtain the list of Spanish provinces whose total population is greater than or equal to 1,900,000 inhabitants and show the total number of inhabitants in each of these provinces. Order this list by decreasing number of inhabitants.

Exercise 21

Obtain an ordered list of the provinces of 'Andalucía' with over one million inhabitants.

Exercise 22 (drop)

Delete the collection of municipalities.

Submit a text file as your solution to this practice session. Transcribe the order used in each step and copy or describe the result obtained in each step.

VNIVERSITAT DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria ETSE-UV

Lab 7: Data Migration and Integration

Objectives

To know and apply good practices in the processes of data migration and data integration from various sources into a database.

Description

This practice involves migrating an existing MySQL database (world1) to a new version of the database using suitable actions for ensuring the quality of the data and, if necessary, redesigning the database. It also involves integrating other data sources, such as data sets available in CSV files (paises_iso) or other tables from a different DBMS such as Oracle (*mundiales*), to enrich the database with new tables or to complete or update existing data.

Submission

You must submit four files:

1) a model or diagram of the final database,

2) a file describing the mapping of tables and fields from the source data to the target,

3) an SQL text file containing all SQL statements in the order in which they were used (except large sets of INSERT) or descriptions of the operation performed when there is no SQL statement (descriptions are included as comments in SQL), and

4) a self-describing SQL file of the final database.

Development

SOURCES

MySQL Database: world1

created in MySQL by executing the wordl1.sql file that creates the world1 schema.

Tables in Oracle: *mundiales* (and olympics for the learning process)

Available in pokemon in the ACD050 schema.

Files from other sources (CSV): paises_iso (and airport_spain for the learning process)

provided as practice material.

PREPARATION

First, we will create the world1 schema from the world1.sql file and add a fake country and a fake city “of our own”, using our surname and first name, e.g. ‘Martinezland’ and ‘José Luis City’.

DATA DISCOVERY / ANALYSIS / NEW DATABASE DESIGN

Identify all the data to be migrated or integrated through analysis and/or “interviews” with the owner/producer of the data (the lecturer), design the new and enriched normalized database, and build a diagram of the database. Write down a mapping table with a format such as the following that includes all fields from all final tables, or at least from the country and *mundiales* final tables:

Source	Source type	Table or file	Field	Data type	Destination Table	Destination field	Observ.
world1	MySQL	country	Code	CHAR(3)	country	Code	
paises_iso	CSV	paises_iso	2	CHAR(52)	country	Nombre	

ETL TO THE MIDDLE WORKING AREA

Migrate the existing world1 database in MySQL to an intermediate workspace in MySQL in a schema named world2_mid:

> export the world1 schema with data to a self-described SQL file, and
>
> modify that file so that the tables and data are created and loaded in a world2_mid schema, and execute the modified file.

Add the fields “include” and “origin” to all the tables created in world2_mid:

> set the fields include=true by default, and
>
> set origin=<”description_of_origin”> according to the origin of each record.

Incorporate data from other databases:

> in world2_mid, create table(s) to incorporate data sources from Oracle table(s), and
>
> export data from Oracle, transform them for loading into MySQL, and execute that load.

Incorporate data from files:

> create the table(s) for the file data (paises_iso.csv), including an automatic IDext, and
>
> execute the load of the data contained in the CSV file(s).

DATA QUALITY CHECK (INTRA-TABLES AND INTER-TABLES)

Detect duplicates and mark them as include=false (with UPDATE).

Detect errors or deficiencies in the tables and correct them with the necessary INSERT or UPDATE.

Check matches and gaps in relationships between tables from different sources.

Correct absences with the necessary INSERT or UPDATE, or mark as include=false.

Perform other suitable checks.

DATABASE RESTRUCTURING

(Lecturer) Modify the city table to add an airport code and airport name to each city from airport_spain

If there is no airport for a city, set NULL values for the code and name of its airport.

If there is only one airport for a city, obtain and set its code and name.

If there is more than one airport for a city, obtain and set the code and name for one of them.

Modify the country table to add fields from paises_iso.

Consider whether a change of the country primary key is possible and desirable.

Modify the *mundiales* table to add the identifiers/codes of the different countries.

Normalize the database and modify the design aspects that are considered appropriate:

(Lecturer) Normalize the language information by creating a language table (id_language, language) and modifying the country_language table to include the related id_language.

Normalize the information about the regions by creating a table region (id_region, region) and modify the table country by adding a field id_region with the corresponding value.

PRE-LOAD PHASE

Create a pre-load space in MySQL in a schema called world2_pre.

Define the tables of the future production system, properly establishing all the primary and foreign keys and all the relevant constraints, and keeping only the fields of interest.

Fill in the target tables in pre-load from the working tables while leaving only the appropriate data.

Redo all the proper quality checks and ensure that all the relevant data have been integrated.

TRANSITION TO PRODUCTION

Export the target tables through a self-described SQL file and modify the table for use in creating and filling the world2_pro schema.