MANIPULANDO DATOS

“Nada es permanente a excepción del cambio. La permanencia es una ilusión de los sentidos”.

(Heráclito)

INSERTAR NUEVOS REGISTROS

La inserción de datos usando SQL es una operación esencial para los sistemas de base de datos relacionales. Cuando su empresa obtiene nuevos datos sobre un cliente, por ejemplo, lo más probable es que una inserción SQL sea la forma en que esos datos ingresan a su base de datos de clientes existente. De hecho, ya sea que lo sepa o no, los datos fluyen hacia las bases de datos utilizando inserciones SQL todo el tiempo. Cuando completa una encuesta de marketing, cuando realiza una compra, cuando se registra a un sistema para poder acceder a través de un login es probable que sus datos se inserten en una base de datos en algún lugar utilizando SQL.

La instrucción para agregar nuevos registros a la tabla se denomina INSERT INTO y es posible escribir la instrucción de dos formas. La primera forma específica tanto los nombres de las columnas como los valores que se insertarán:

```
insert into <nombre_tabla> (column1, column2, column3,...columnN)
values (value1, value2, value3,...valueN);
```

Para esta instrucción debemos colocar primero el nombre de la tabla donde se incorporará el registro seguido de la lista de campos (separados por comas) donde se insertarán dichos valores.

La segunda forma se usa en el caso de que se agreguen valores para todas las columnas de la tabla, en ese caso no es necesario que especifique los nombres de las columnas en la consulta SQL. Sin embargo, debemos asegurarnos de que el orden de los valores esté en el mismo orden que las columnas de la tabla. En este caso la sintaxis INSERT INTO sería más simple:

```
insert into <nombre_tabla>
values (value1, value2, value3,...valueN);
```

Por ejemplo, insertaremos un nuevo empleado en la tabla employees. Como para poder insertar un empleado necesitamos de su ´puesto laboral primero agregaremos el puesto laboral que no existe en la tabla Jobs:

```
insert into jobs(job_id,job_title,min_salary,max_salary) values ('DBA','Admin
DBA',6000.00,12000.00);
```

Ingresado el puesto laboral ahora ingresaremos el empleado asociado a ese nuevo puesto:

```
insert into employees values
(207,'Dolores','Garcia',DGARCIA',
'1134190112',TO_DATE('04/01/2020','DD/MM/YYYY') ,20,10000,101,6);
```

Para insertar múltiples filas en Oracle 23ai puede realizarlo de la siguiente manera:

```
insert into regions (region_id,region_name) values (5,'Australia y
Oceanía'),(6,'Europa del Este');
```

Fíjese en la coma para separar el conjunto de datos a insertar. Al momento vera el mensaje:

```
2 filas insertadas.
```

MODIFICACION DE DATOS

Para modificar los valores de las columnas de los registros existentes debe usar el comando UPDATE. Este comando tiene tres componentes fundamentales: el nombre de la tabla, el cambio que desea aplicar y los registros donde se aplicaran los cambios o las modificaciones:

```
update <nombre_tabla> set
<columnl>=<value1>,<column2>=<value2>,…,<column_n>=<value_n>
where <condition that uniquely identifies the record that needs to be update>;
```

Para este ejemplo actualizaremos el salario del empleado cuyo código es 145 en un 3%:

```
update employees
set salary=salary+(salary*0.03)
where employee_id=145;
```

También podemos usar subconsultas para actualizar las filas de una tabla. En este ejemplo volveremos a actualizar el empleado cuyo id es 145 modificando su salario con el salario más alto encontrado en la tabla empleados:

```
update employees employees
set salary=(select max(salary) from employees)
where employee_id=145;
```

Si deseamos actualizar todos los datos la clausula WHERE no iría, veamos el siguiente ejemplo:

```
update employees set email=CONCAT(email,'@mycompany.com');
```

La nueva versión de Oracle permite también la actualización de filas usando JOINS directos usando la clausula FROM: El beneficio de esto es hacer no solo que el código sea más corto, sino que también sea más legible para los desarrolladores de SQL. Por ejemplo, supongamos que queremos aumentar los salarios de los empleados que trabajan en finanzas, en una primera consulta podríamos obtener cuales son esos empleados y que salarios tienen:

```
select e.employee_id, e.salary from employees e, departments d where
e.department_id=d.department_id AND d.department_name='Finance';
```

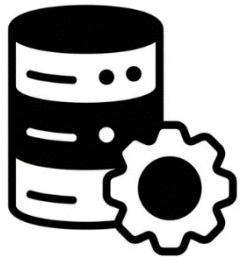
En Oracle 23ai podemos realizar lo siguiente:

```
UPDATE employees e SET e.salary=e.salary*1.05 FROM departments d WHERE
e.department_id=d.department_id AND d.department_name='Finance';
```

ELIMINACION DE REGISTROS

Para eliminar registros se usa el comando DELETE. La palabra clave FROM indica en que tabla se procederá a la eliminación de filas. La cláusula where indica cual es el criterio a aplicar para la eliminación de registros.

```
delete from <nombre_tabla>
where <condition that uniquely identifies the record that needs to be update>;
```



Nota

La cláusula WHERE es opcional. Si no se proporciona esta cláusula, se eliminará todos los registros de la tabla.

Para este ejemplo eliminaremos la región cuyo id es la numero 6.

```
delete from regions where region_id=6
```

Como este comando es tal vez uno de los más peligrosos podemos usar un método antes de eliminar los datos realmente. Para ello sustituimos la palabra reservada DELETE por la sentencia SELECT de esta forma:

```
select * from regions where region_id=6
```

De esta forma tendremos los registros correctos que luego se eliminaran con la instrucción DELETE.
Así como Oracle 23ai permite JOINS en los updates también podemos realizar DELETES con JOINS incluidos. Por ejemplo, supongamos que queremos eliminar todo el historial de trabajo del empleado cuyo correo electrónico es JTAYLOR, para ello podemos escribir la siguiente consulta:

```
delete job_history jh from employees e where jh.employee_id=e.employee_id and
e.email='JTAYLOR'
```

CONCEPTOS DE NORMALIZACIÓN

La normalización es el proceso de organizar datos en una base de datos. Esto incluye la creación de tablas y el establecimiento de relaciones entre esas tablas de acuerdo con las reglas diseñadas tanto para proteger los datos como para que la base de datos sea más flexible mediante la eliminación de la redundancia y las dependencias incoherentes. Los datos redundantes desperdician espacio en disco y aumentan la probabilidad de que se produzcan errores e incoherencias. El segundo principio es que es importante que la información sea correcta y completa. Si la base de datos contiene información incorrecta, los informes que recogen información de la base de datos contendrán también información incorrecta y, por tanto, las decisiones que tome a partir de esos informes estarán mal fundamentadas. Para eliminar estos problemas existen las Formas Normales. Existen básicamente tres formas Normales (FN). Las reglas para la forma normal son acumulativas. Es decir, para que una entidad cumpla las reglas de la segunda forma normal, también debe cumplir las reglas de la primera forma normal. Aunque son posibles otros niveles de normalización, la tercera forma normal se considera el nivel más alto necesario para la mayoría de las aplicaciones.

Al igual que con muchas reglas y especificaciones formales, los escenarios del mundo real no siempre permiten el cumplimiento perfecto. En general, la normalización requiere tablas adicionales y algunos clientes lo consideran engorrosos. Si decide infringir una de las tres primeras reglas de normalización, asegúrese de que la aplicación anticipa los problemas que puedan producirse, como datos redundantes y dependencias incoherentes.

PRIMERA FORMA NORMAL (1FN)

Al aplicar el proceso de normalización en una relación, se empieza por comprobar si la relación está en primera forma normal y, si no es el caso, se efectúan las modificaciones oportunas para conseguirlo. Para que una base de datos esté en primera forma normal (1FN) cada columna de una tabla debe ser atómica, es decir, que cada atributo debe contener un único valor del dominio. Por ejemplo, supongamos que una nuestra empresa necesita capacitar a nuestros empleados con distintos cursos y formadores. Esta podría ser la situación inicial:

ID_Curso	Nombre_Curso	Apellido_Empleado	Instructor_ID	Nombre_Instructor	Fecha_Inicio_Curso
1	Economía	Ford,Roshak, Lee	1	Chris	1/08/25
1	Economía	Myrick,Singh, Lapp,Baer	1	Chris	12/11/25

Los problemas típicos de esta tabla no normalizada incluyen:

- Repetición de la información sobre los cursos cada vez que un empleado se inscribe.
- Repetición de información de los instructores.
- Información de empleados duplicados si cada uno decide tomar varios cursos.

Como se ve, esta relación no está en primera forma normal (1FN) ya que el atributo apellido del empleado es multivaluado. Podríamos modificar la entidad separando los nombres de los empleados en distintas columnas:

ID_Curso	Nombre_Curso	Apellido#1	Apellido#2	Apellido#3	Apellido#4	Instructor_ID	Nombre_Instructor	Fecha_Inicio_Curso
1	Economía	Ford	Roshak	Lee		1	Chris	1/08/25
1	Economía	Myrick	Singh	Lapp	Baer	1	Chris	12/11/25

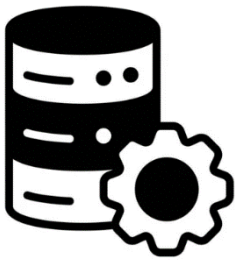
Ahora pregúntese: ¿Qué sucedería si un empleado se anota en el curso de economía que comienza el 12/11/25?. Agregar una nueva columna denominado apellido#5 no es la respuesta; requiere modificaciones en la tabla y no admite de forma fluida un número dinámico de empleados. ¿Qué pasaría si agregamos un nuevo curso donde solo se anota un solo empleado? ¿Cuántos atributos quedarían vacíos? Por lo tanto, no use varios campos en una misma tabla para almacenar datos similares. Para transformar una tabla en una normalizada a la forma 1FN, tenemos que garantizar que haya un único valor en la intersección de cada fila y columna. Esto se consigue eliminando el grupo repetitivo. En su lugar, inserte toda la información de los empleados en una tabla separada llamada empleados y, a continuación, vincule la tabla empleados a capacitaciones.

SEGUNDA FORMA NORMAL (2FN)

Una tabla está en segunda forma normal (2FN) cuando está en 1FN y además todos los atributos que no forman parte de la clave principal tienen dependencia funcional de la clave completa y no de parte de ella. Las reglas definidas nos plantean las siguientes dudas: ¿Qué es una clave principal? ¿Qué es una dependencia funcional? Una clave principal es un conjunto de una o más columnas que identifican de manera única (no repetida) a una fila. Una Dependencia Funcional es una relación de implicancia entre 2 columnas, se dice que un atributo tiene dependencia funcional de otro cuando a cada valor del primero le corresponde un solo valor del segundo. Ejemplo:

ID_Curso	Nombre_Curso
1	Economía
2	Base de datos

En este ejemplo, si conocemos el Id del Curso, podemos obtener el nombre del curso. Con esto, podemos decir que el nombre del curso depende funcionalmente del Id del Curso. Es decir que para cada código de curso solo puede haber un nombre de curso. La Dependencia funcional completa es cuando un atributo depende de otro que es un atributo compuesto (un conjunto de atributos), se dice que la dependencia funcional es completa si el atributo dependiente no depende de ningún subconjunto del atributo compuesto. Supongamos que tenemos la clave principal compuesta por Código de Curso + Código de Instructor + Fecha de Inicio del curso (como muestra la primera figura de la 1FN), la descripción del curso depende de un subconjunto de la clave. No de toda la clave. Por lo tanto, no hay dependencia funcional completa.



Nota

La dependencia funcional es un concepto en las bases de datos que describe la relación entre atributos. Si A y B son subconjuntos de atributos de una tabla, diremos que B depende funcionalmente de A (o también que A determina a B) si cada valor de A tiene asociado siempre un único valor de B.
*A → B (**se lee: A determina a B**)*

Cabe destacar que, si la clave principal no es una clave compuesta, todos los atributos no clave siempre dependen funcionalmente de la clave principal. Una tabla en primera forma normal que contiene solo un campo como clave principal se encuentra automáticamente en segunda forma normal.

TERCERA FORMA NORMAL (3FN)

Una base de datos está en tercera forma normal (3FN) si está en 2FN y no existen atributos que no pertenezcan a la clave primaria que puedan ser conocidos mediante otro atributo que no forme parte de la clave primaria. Es decir, que no existan dependencias funcionales transitivas. Veamos el siguiente ejemplo de la tabla cursos:

ID_Curso	Nombre_Curso	Instructor_ID	Nombre_Instructor
1	Economía	1	Chris
2	Base de datos	2	Natalia

Supongamos que la clave principal en esta entidad es el código del curso ya que define de manera única a cada fila. Pero que sucede cuando trabajamos con dependencias funcionales, el nombre del instructor depende del código del instructor (no del código del curso). Es decir, si modificamos el nombre del instructor debería cambiar el código de instructor. Por ello debemos mover dichas columnas y formar una nueva entidad.

CREAR UNA TABLA

Para crear tablas en SQL se usa la instrucción create table.

```
create table <nombre_tabla>;
```

Pero antes de trabajar con esta instrucción veremos algunas pequeñas cosas para el armado de nuestra tabla.

TIPOS BASICOS DE DATOS EN ORACLE

En Oracle, cada valor tiene un tipo de datos que identifica cómo Oracle almacenará y procesará los datos. Entonces, cuando crea una tabla, debe asignar un tipo de datos para cada columna. Un tipo de datos define lo siguiente:

- Qué valores se pueden asignar para el tipo especificado.
- Cómo se pueden almacenar.
- Qué propiedades se pueden asociar con el valor.
- Qué operaciones se pueden realizar sobre el valor.

TIPOS DE DATOS

En Oracle, los tipos de datos de caracteres almacenan cadenas de letras, símbolos y números Oracle también admite el tipo de dato numérico que puede almacenar valores numéricos fijos o flotantes. El tipo de dato fecha almacena fechas y horas. Finalmente podemos guardar datos no estructurados como fotos e ilustraciones con el tipo de dato blob.

- **CHAR:** Almacena un valor de cadena de longitud fija de 1 a 2000 bytes. Si la cadena es menor que la longitud definida se rellena con espacios en blanco a la derecha. Se almacena un byte en memoria por cada caracter. Ejemplo: char(10) si deseamos almacenar el conjunto 'abcde' veremos que rellenara con espacios en blanco hasta completar la longitud de 10 caracteres. Si no se especifica un valor por defecto es 1.

- **VARCHAR2:** Es el reemplazo de varchar. Almacena un valor de cadena de longitud variable de como máximo 32 767 bytes en Oracle 12.

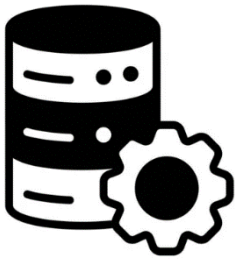
- **NUMBER:** Almacena todo tipo de números, incluidos números enteros y números decimales. La precisión es el número total de dígitos de un número. La escala es el número de dígitos situados a la derecha de la coma decimal de un número. Por ejemplo, el número 123.45 tiene una precisión de 5 y una escala de 2. En Oracle un number tiene una precisión que va de 1 a 38. Para el ejemplo anterior debemos escribir number(5,2). Si se desea almacenar el número 77777,77, se mostrará el mensaje de error porque el valor es mayor que la precisión especificada. Para definir un número entero, utilice la siguiente forma number(p) donde p es la precisión.

- **DATE:** El tipo de datos DATE almacena valores de fecha y hora en una columna de tabla. Incluye un año, un mes, un día, horas, minutos y segundos.

- **BLOB:** El tipo de datos BLOB (Binary Large Object) almacena un archivo de datos binarios no estructurados con un tamaño máximo de hasta 128 TB.

- **JSON:** Oracle 23ai ofrece el nuevo tipo de datos nativo denominado JSON (JavaScript Object Notation). Aunque JSON tiene sus raíces en JavaScript, es uno de los formatos de datos más populares basado en texto para almacenar e intercambiar datos a través de diferentes lenguajes de programación y plataformas.

- **BOOLEAN:** Oracle Database 23ai ahora admite el tipo de dato BOOLEAN, compatible con el estándar ISO SQL Esto permite almacenar valores TRUE (Verdadero) y FALSE (Falso) en tablas.



Nota

Tenga en cuenta que el tipo de datos VARCHAR no debe usarse porque su semántica podría modificarse en un futuro máximo. En su lugar, debe utilizar VARCHAR2.

MANEJO DE JSON IN ORACLE

Tradicionalmente, el modelo relacional ha sido valorado por su eficiencia y la integridad de los datos que ofrece a través de la normalización. No obstante, su estructura basada en tablas, filas y columnas puede dificultar el desarrollo de aplicaciones, ya que suele requerir operaciones complejas para acceder y manipular la información. JSON (acrónimo de JavaScript Object Notation) es un formato basado en texto que se utiliza para almacenar, transferir e intercambiar datos en aplicaciones y servicios modernos. Aunque JSON tiene sus raíces en JavaScript, se ha convertido en un formato de datos que simplifica el intercambio de datos entre diversas plataformas y lenguajes de programación. Si está trabajando en desarrollo web, análisis de datos o ingeniería de software, JSON es un formato de datos importante que debería comprender. Entonces la pregunta sería: ¿En qué me beneficia almacenar datos en dicho formato en Oracle? Una de las ventajas es contar con una estructura de datos flexible que no se encuentra con restricciones del modelo relacional. Otro de los beneficios es que muchas empresas usan APIs tipo REST o RESTful para comunicarse con aplicaciones internas o de terceros para llevar a cabo tareas. En otras palabras, las API REST se comunican con un servidor apoyándose en el protocolo HTTP para obtener datos o ejecutar una función, de manera que el sistema comprenda la solicitud y devuelva una respuesta. El tema aquí es que la respuesta generalmente viene en formato JSON. Simplicidad, flexibilidad y velocidad son las características de este formato de archivos. Muchos sistemas de base de datos actuales usan este formato para almacenar datos (como los sistemas de bases de datos NoSQL basado en documentos), en este caso los datos se almacenan en documentos en lugar de la forma columna/fila con lo que estamos familiarizados en base de datos relacionales. La sintaxis de un archivo JSON es la siguiente:

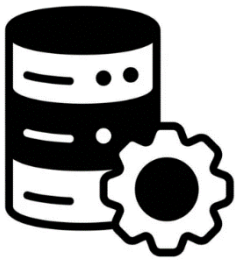
- Los datos JSON se escriben como pares nombre/valor. Un par nombre/valor consta de un nombre de campo entre comillas dobles, seguido de dos puntos y un valor que pude ser del tipo string (alfanuméricos), number(números), boolean (booleanos) ,null (nulos), object (objetos), array (arreglos); estos dos últimos son los que hacen que JSON sea flexible.

- Los objetos JSON se escriben entre llaves { }..

- Los arreglos o arrays en JSON están escritos entre corchetes [ ].. Un arreglo puede contener objetos.

Supongamos que queramos almacenar la información de una serie de Netflix en un archivo JSON, una posibilidad sería la siguiente:

```
{
    "series_id": "12345",
    "title": "Stranger Things",
    "description": "Cuando un niño desaparece, un pequeño pueblo descubre un
     misterio que involucra experimentos secretos y fuerzas sobrenaturales.",
    "release_year": 2016,
    "genres": [
        "Drama",
        "Horror",
        "Sci-Fi"
    ]
}
```

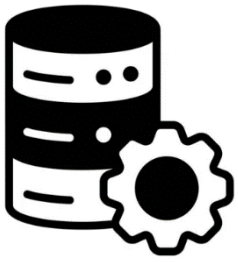


Nota

Se puede validar un documento JSON usando el sitio: https://jsonlint.com/, este sitio expondrá los posibles errores en la construcción de un archivo JSON. Otro sitio interesante es, https://jsoncrack.com/ que permite no solo editar un archivo JSON sino que también permite visualizar en un tipo de grafico el contenido del mismo.

ÍNDICES

Los índices en SQL son estructuras especiales utilizadas para mejorar la eficiencia y velocidad de las consultas en bases de datos. Un índice es una tabla auxiliar que contiene una copia selectiva de datos de una columna o un conjunto de columnas de la tabla principal. ¿Por qué utilizar índices? Porque sin ellos usted tiene que realizar una lectura completa en cada tabla. Basta pensar en las viejas guías telefónica: ellas están indexadas por su nombre, así que si se le pide que encuentre todos los números de teléfono de personas cuyo apellido es Diaz, puede hacerlo en forma rápida. Sin embargo, si le pido que encuentre a todas las personas que tienen un número de teléfono que comienza con 1588, no tendrá más remedio que leer la guía telefónica completo. Lo mismo ocurre con las tablas de la base de datos, si busca algo en filas no indexadas deberá escanear toda la tabla.



Nota

Los índices de bases de datos juegan un papel crucial en el mundo de las bases de datos Oracle. A medida que aumentan los volúmenes de datos, los usuarios de bases de datos suelen tener que lidiar con velocidades de consulta lentas.

¿Podemos poner índices en todas partes? No. Los índices vienen con costos. Deben crearse sólo cuando sea necesario:

- Los índices ocupan espacio adicional en el disco.
- A medida que los datos se insertan, actualizan o eliminan de la tabla, los índices también deben actualizarse, lo que puede implicar cierto costo en términos de rendimiento en las operaciones de escritura.
- Un mal diseño de índices o la creación de índices innecesarios puede conducir a un rendimiento más lento en lugar de mejorarlo.
- No indexe en tablas pequeñas: Evite crear índices en tablas pequeñas con pocos registros, ya que puede ser más costoso mantener los índices que la mejora en el rendimiento que aportan.
- Indexe las columnas utilizadas en las cláusulas JOIN y WHERE: Es fundamental indexar las columnas utilizadas en cláusulas JOIN y WHERE, ya que esto acelera el proceso de búsqueda y filtrado de registros.

Para crear un índice necesita usar la sentencia:

```
create index <index_name>
on <table_name> (column1, column2, ...columnN);
```

Esta declaración es la forma más simple de sintaxis. Donde <index_name>, es el nombre del índice que se creará, <table_name> es el nombre de la tabla y column1, column2,… columnN las columnas de la tabla que se incluirán en ese índice. El comando DROP INDEX en Oracle permite a los usuarios eliminar cualquier índice existente del esquema de base de datos actual. Esto no afectará físicamente a la tabla porque los índices son objetos independientes y se almacenan por separado. La sentencia para eliminar el índice de Oracle es la siguiente:

```
drop index <index_name>;
```

Donde <index_name> es el nombre del índice.

CONSTRAINTS EN SQL

En términos simples, las constraints o restricciones son reglas que establecemos para limitar el tipo de datos o los valores que pueden ingresar o modificarse en una tabla. Si hay alguna violación de la restricción el motor abortara la acción. Las constraints más comunes en SQL son PRIMARY KEY, FOREIGN KEY, NOT NULL, IDENTITY, UNIQUE y CHECK.

CLAVE PRINCIPAL y CLAVES FORANEAS

La clave principal es un índice único especial. Solo se puede definir un índice de clave principal en una tabla. La clave principal se utiliza para identificar de forma única un registro y se crea utilizando la palabra clave PRIMARY KEY. Puede especificar una clave principal en una tabla de dos maneras:

1) Junto al tipo de datos de la columna cuando se la define.
2) Al final de todas las declaraciones de columna

El método que elija dependerá de los estándares de su equipo de desarrollo. Para declarar una clave principal en la misma línea que la definición de la columna se debe escribir lo siguiente:

```
create table <nombre_tabla> (
  column_name data_type PRIMARY KEY,
  ...
);
```

El método fuera de la definición de la columna es poco distinto al anterior:

```
create table <nombre_tabla> (
  column_name data_type,
  ...
  CONSTRAINT pk_name PRIMARY KEY (column_name)
);
```

Esta definición se debe agregar al final de la especificación de todas las columnas de la tabla y debe comenzar con la palabra CONSTRAINT. El nombre de la clave es necesario para este punto puede colocar un nombre que identifique a la clave principal de dicha tabla. Dentro de los paréntesis después de la palabra PRIMARY KEY debe especificar los nombres de las columnas. Si hay más de una debe separarlos con una coma.
La clave externa o FOREIGN KEY, es una columna o varias columnas, que sirven para señalar cual es la clave primaria de otra tabla. La columna o columnas señaladas como FOREIGN KEY, solo podrán tener valores que ya existan en la PRIMARY KEY de la otra tabla. La integridad referencial asegura que se mantengan las referencias entre las claves primarias y las externas. También controla que no pueda eliminarse un registro de una tabla ni modificar la clave primaria si una clave foránea o externa hace referencia al registro.

```
create table <nombre_tabla> (
  column_name data_type,
  ...
  CONSTRAINT fk_nametbl1_nametbl2 FOREIGN KEY (this_tables_column)
  REFERENCES other_table_name (other_column_name)
);
```

Para referirnos a una clave foránea debemos comenzar con la palabra reservada CONSTRAINT, luego deberemos colocar el nombre de la clave foránea. El nombre debe ser único en toda la base de datos, por lo que se recomienda comenzar con el prefijo fk, luego debemos especificar las palabras claves FOREIGN KEY, con el nombre de la tabla y la referencias a la tabla y columna donde encontraremos la clave principal de la tabla externa.

OTRAS RESTRICCIONES

La restricción NOT NULL define que la columna no deberá aceptar valores nulos. UNIQUE indica que el valor debe ser único no puede haber dos valores iguales. Las restricciones CHECK se usan para comprobar ciertos tipos de valores. Puede crear una restricción CHECK con cualquier expresión lógica (booleana) que devuelva TRUE (verdadero) o FALSE (falso), Por ejemplo:

```
constraint chk_salary_min CHECK(min_salary> 0)
```

La definición de restricción comienza con la palabra clave CONSTRAINT. Esta restricción lleva un nombre que debe ser único dentro del esquema. La definición de la restricción de verificación consiste en la palabra clave CHECK seguida de una expresión entre paréntesis. En este caso la expresión a evaluar verifica si el salario mínimo es mayor a cero. Otro tipo de restricción que se puede incorporar en ORACLE 12 es la columna IDENTITY, este tipo de restricción permite colocar a una columna como incremento automático. Es decir, cuando insertemos nuevos registros y se refiere a dicha columna no será necesario ingresar un valor ya que le motor lo incorporará automáticamente.
Finalmente, con todo lo visto y aplicando los conceptos de normalización haremos una modificación sobre nuestro sistema de RRHH para ello almacenaremos los datos de las capacitaciones que realicen nuestros empleados. Para ello se crearán tres tablas más, una denominada COURSES con todos los cursos de capacitación que ofrece nuestra empresa, la tabla nexo entre empleados y cursos denominada EMPLOYEES_COURSES que contendrá las capacitaciones realizadas por los empleados y la tabla INSTRUCTORS que contiene la información de los instructores de los cursos. Para crear la tabla instructores debería escribir lo siguiente:

```
create table instructors (
    instructor_id NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY PRIMARY KEY,
    data JSON
);
```

El campo data es un tipo de JSON que permitirá almacenar datos de los instructores, para introducir dos registros completos con datos usaremos la instrucción INSERT de esta forma:

```
insert into instructors (data) values ('{
    "first_name": "David",
    "last_name": "Lawson",
    "email": "david.lawson@myemail.com",
    "skills": [
        "Inteligencia Artificial",
        "Machine Learning",
        "Big Data",
        "Cloud Computing"
    ]
}'),
('{
    "first_name": "Diana",
    "last_name": "Gray",
    "email": "diana.gray@myemail.com",
    "skills": [
        "Economía",
        "Estadística",
        "Derecho"
    ]
}');
```

Ahora visualizaremos los datos del JSON recién ingresado:

```
select data from instructors;
```

Se puede tener una salida más bonita usando la función JSON_SERIALIZE en el siguiente comando:

```
select json_serialize(i.data PRETTY) from instructors i;
```

Podemos visualizar solo los apellidos con el siguiente comando SELECT:

```
select i.data.last_name from instructors i;
```

Si quisiéramos ver solo la información de Diana Gray deberíamos escribir la siguiente instrucción:

```
select json_serialize(i.data PRETTY) FROM instructors i where
json_value(i.data, '$.last_name') = 'Gray';
```

El próximo paso será crear la tabla courses:

```
create table courses (
    course_id NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY,
    course_name VARCHAR2(100),
    start_date DATE NOT NULL,
    end_date DATE,
    instructor_id NUMBER,
    CONSTRAINT fk_instructor_id FOREIGN KEY (instructor_id)
    REFERENCES instructors (instructor_id),
    CONSTRAINT pk_course_id PRIMARY KEY (course_id),
    CONSTRAINT chk_dates CHECK (start_date <= end_date)
);
```

MODIFICAR LA DEFINICION DE LA COLUMNA

Mediante el comando ALTER TABLE, puede modificar una definición de una columna de una tabla existente. Por ejemplo, supongamos que quiero agregar que el nombre del curso no pueda contener un NULL. El comando sería el siguiente:

```
alter table courses MODIFY course_name varchar2(100) NOT NULL;
```

De esta forma estaría agregando una restricción NOT NULL al campo course_name de la tabla COURSES. Podemos usarla también para modificar la longitud del campo course_name:

```
alter table courses MODIFY course_name varchar2(50);
```

También podemos usar la instrucción para colocar un valor por defecto. Un valor por default (o por omisión) es aquel valor que se le dará a un campo si al momento de insertar su registro no se le da un valor específico. Por ejemplo:

```
alter table courses MODIFY start_date date default sysdate;
```

En este caso estamos modificando la definición de la columna start_date desde estos momentos tendrá el valor por defecto la fecha de hoy. Por eso cuando insertemos el próximo registro al omitir este campo almacenara la fecha actual.

Si ahora realiza la siguiente consulta:

```
describe courses
```

Podrá observar que las modificaciones realizadas en la estructura de la tabla COURSES. También podemos agregar una nueva columna para ver la dificultad del curso:

```
alter table courses ADD difficultylevel varchar2(20);
```

En este caso agregamos la columna difficultylevel con un tipo de dato alfanumérico de 20 caracteres como máximo. El próximo comando SQL agregará una restricción permitiendo que solo se ingrese los valores low, médium y high en el campo recién agregado.

```
alter table courses ADD CONSTRAINT chk_difficultylevel CHECK (difficultylevel
in ('low','medium','high'));
```

También crearemos in índice (aunque para tan pocos registros no es necesario, pero nos sirve como practica) denominado idx_course sobre la columna course_name de la tabla courses:

```
create index idx_course_name on courses (course_name);
```

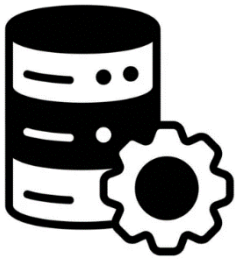
Podemos listar los índices usando el comando:

```
select * from user_indexes where table_name='COURSES';
```

Si ahora realiza la siguiente consulta:

```
describe courses
```

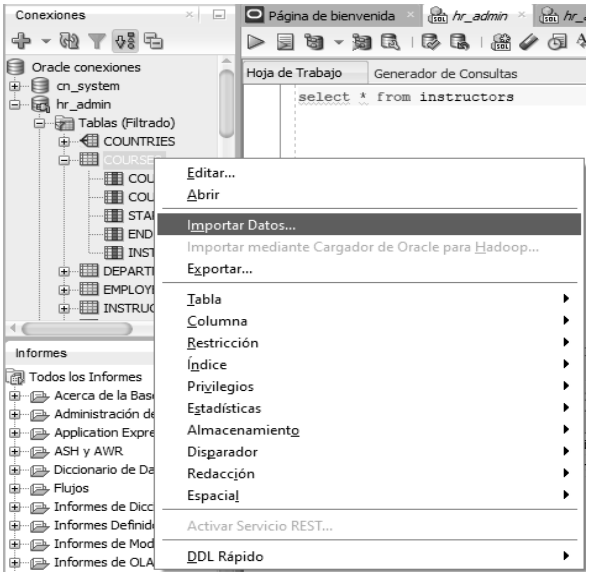
Podemos ver las últimas modificaciones realizadas. Ahora introduciremos datos que están almacenados en un archivo csv. Un archivo CSV (Valores Separados por Comas) es un archivo con datos en texto plano donde tendremos un listado con la información para insertar dentro de la tabla:



Nota

Puede eliminar una columna utilizando el comando ALTER TABLE <nombre_tabla> DROP COLUMN <nombre_columna>

Sobre nuestra conexión ubicada en la tabla Courses seleccionaremos con el botón derecho, la opción IMPORTAR DATOS...



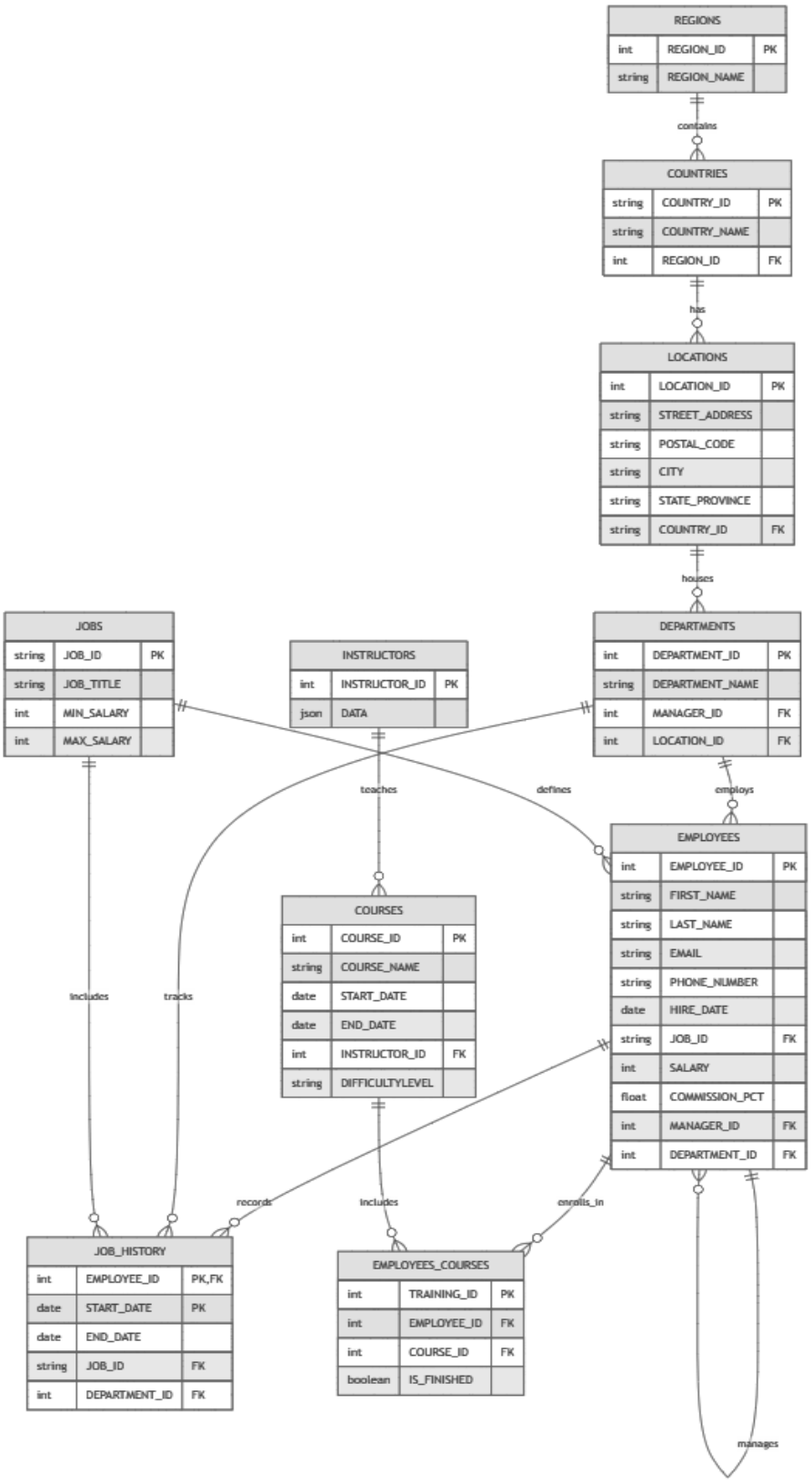
Luego en la nueva venta seleccionaremos el archivo CSV. Al momento tendremos una previsualización mostrando el contenido del archivo. En la siguiente ventana en Métodos de Importación seleccionaremos la opción INSERTAR. En el paso siguiente elegiremos las columnas que formaran parte de la inserción. En la ventana siguiente podremos ver las columnas con los valores a insertar para ver si tendremos algunos problemas en la inserción. En la ultima ventana seleccionaremos la opción Finalizar. Al momento habremos insertado todos los registros del archivo. Ahora crearemos la tercer tabla denominada EMPLOYEES_COURSES:

```
create table employees_courses (
    training_id NUMBER GENERATED BY DEFAULT ON NULL AS IDENTITY PRIMARY KEY,
    employee_id NUMBER NOT NULL,
    course_id NUMBER NOT NULL,
    is_finished BOOLEAN NOT NULL,
    CONSTRAINT fk_employee_id FOREIGN KEY (employee_id)
    REFERENCES employees (employee_id),
    CONSTRAINT fk_course_id FOREIGN KEY (course_id)
    REFERENCES courses (course_id)
);
```

Ahora podemos insertar la capacitación de un empleado:

```
insert into employees_courses values ('',105,5,FALSE);
```

Finalmente podemos ver nuestro nuevo DER:



EXPRESION CASE

La expresión CASE le permite evaluar una lista de condiciones y devuelve uno de los múltiples resultados. Puede usar la expresión CASE en SELECT,UPDATE, DELETE y en las cláusulas WHERE, HAVING y ORDDER BY. La sintaxis de la expresión CASE es la siguiente:

```
case [expression]
  when condition_1 then result_1
  when condition_2 then result_2 ...
  when condition_n then result_n
  else result
end
```

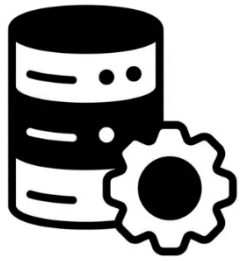
La expresión CASE se evalúa de arriba a abajo. Si una condición es verdadera, entonces se ejecuta la cláusula THEN correspondiente y la ejecución salta inmediatamente a la cláusula END. En el caso de que ninguna cláusula WHEN cumpla las condiciones especificadas, entonces la consulta ejecutará el código indicado en la sentencia ELSE. Si no se incluye la sentencia ELSE y no se cumple ninguna condición WHEN, entonces la base de datos devolverá un NULL como resultado. Para ello resolveremos una nueva consulta en ella evaluaremos el valor del salario:

```
select first_name, last_name, salary,
case
  when salary<6000 THEN 'Salario Bajo'
  when salary>=6000 AND salary<10000 THEN 'Salario Medio'
  when salary>=10000 THEN 'Salario Alto'
end
from employees order by first_name
```

Si el salario es menor a 6000 se creará una nueva columna con el texto Salario Bajo. Si el valor se encuentra entre 6000 y es menor de 10000 el texto que aparecerá será Salario Medio. Si el salario es mayor a 10000 el texto que finalmente que aparecerá será Salario Alto. El valor que aparece en la sentencia ELSE es cuando no cumple con ninguna de la sentencias WHEN..THEN. Acá podemos ver un ejemplo:

```
select course_name,difficultylevel,
(case
  difficultylevel
  when 'low' THEN 'Sin requisitos'
  else 'Con requisitos'
end) "Requisitos"
from courses
```

En este caso sino cumple con ningún WHEN…THEN el valor devuelto es: Con requisitos



Nota

Para desarrolladores experimentados el uso de la expresión CASE es similar a un IF de un lenguaje de programación de alto nivel como (C, Java, PHP etc.). Una declaración IF le permite hacer algo si una condición es verdadera y otra cosa si la condición es falsa.

A continuación, resolveremos un ejemplo más:

```
select
    e.employee_id,
    e.first_name || ' ' || e.last_name AS empleado,
    TO_CHAR(e.hire_date, 'YYYY') AS año_contratación,
    case
        when MONTHS_BETWEEN(SYSDATE, e.hire_date)/12 < 5 then 'Novato'
        when MONTHS_BETWEEN(SYSDATE, e.hire_date)/12 < 10 then 'Semi senior'
        when MONTHS_BETWEEN(SYSDATE, e.hire_date)/12 < 15 then 'Senior'
    end,
    d.department_name
from employees e inner join departments d
on e.department_id=d.department_id
order by e.first_name;
```

En este ejemplo calcularemos la antigüedad de cada empleado en la organización, para cada empleado informaremos si es Novato, Semi senior o Senior en base de su fecha de contratación. También obtendremos el nombre de departamento de cada empleado que se encuentra en la tabla departamentos.

FUNCIONES ANALITICAS

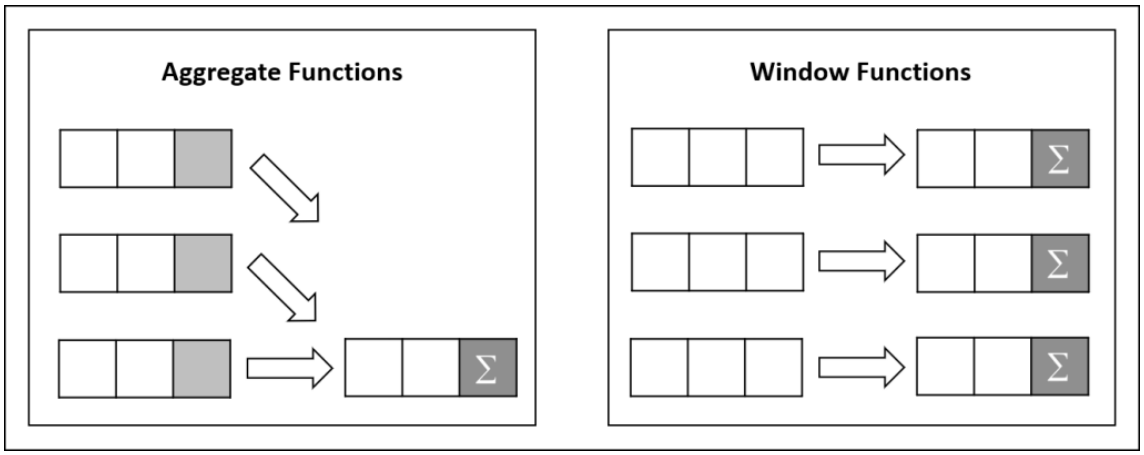
Probablemente la forma más fácil de comprender las funciones analíticas es comenzar observando las funciones agregadas. Una función agregada, como sugiere el nombre, agrega datos de varias filas en una sola fila de resultados. Por ejemplo, podríamos usar la función agregada AVG () para obtener un promedio de todos los salarios de los empleados en la tabla employees:

```
select
    AVG(salary)
from
    employees
```

La clausula GROUP BY nos permite aplicar funciones agregadas a subconjuntos de filas. Por ejemplo, podríamos querer mostrar el salario promedio de cada departamento:

```
select department_id, AVG(salary)
from   employees
group by department_id
order by department_id;
```

En ambos casos, la función de agregado reduce el número de filas devueltas por la consulta. Las funciones analíticas también operan en subconjuntos de filas, de forma similar a las funciones agregadas con un GROUP BY, pero no reducen el número de filas devueltas por la consulta.



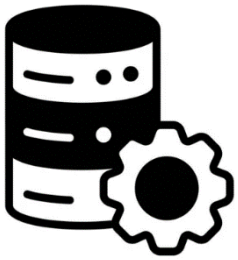
Por ejemplo, la siguiente consulta informa el salario de cada empleado, junto con el salario promedio de los empleados dentro del departamento.

```
select
    employee_id,
    first_name,
    department_id,
    salary,
    AVG(salary)
    OVER(PARTITION BY department_id) "Promedio Salario"
from
    employees;
```

Esta vez la función AVG es una función analítica, que opera en el grupo de filas definidas por el contenido de la cláusula OVER. Este grupo de filas se conoce como ventana, razón por la cual las funciones analíticas a veces se denominan funciones ventana (window functions). Observe cómo la función AVG aún informa el promedio de cada departamento, como lo hizo en la consulta GROUP BY, pero el resultado está presente en cada fila, en lugar de reducir el número total de filas devueltas. Esto se debe a que las funciones analíticas se realizan en un conjunto de resultados después de que se completan todas las cláusulas JOIN, WHERE y GROUP BY, pero antes de que se realicen la operación HAVING y ORDER BY.

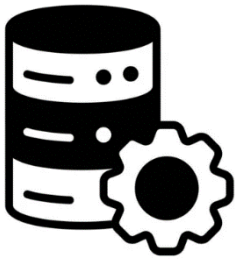
La cláusula PARTITION BY junto a la cláusula OVER se utiliza para dividir el conjunto de resultados de la consulta en subconjuntos de datos o particiones. Las funciones analíticas que se utiliza se aplican a cada partición por separado y el cálculo que realiza la función se reinicia para cada partición. La cláusula OVER indica que se deben dividir los resultados por department_id, a su vez creamos una columna denominada Promedio Salario y en cada fila se calcula el promedio del salario de cada department_id que a diferencia de la cláusula GROUP BY el promedio aparece por cada uno de los empleados repitiéndose en caso de igualdad de código de departamento. Aprender la sintaxis de una función analítica es la mitad de la batalla para aprovechar su poder para un procesamiento de consultas eficiente. La sintaxis para las consultas analíticas es:

```
FUNCTION_NAME( column | expression,column | expression,... )
OVER ( Order-by-Clause )
```



Nota

Imagínese el caso de un Analista de Datos que quiera saber el promedio de salario por cada departamento para predecir los salarios a futuro a través de una variable económica como el nivel de inflación de un país.

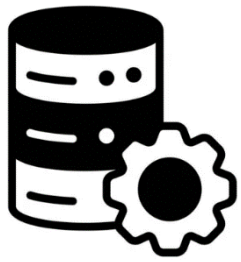


Nota

Si no se especifica la cláusula PARTITION, la agregación se realiza en todos los registros de la tabla.

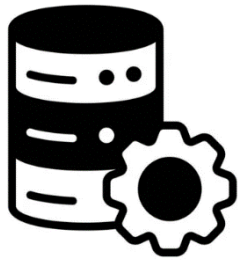
Otra necesidad del negocio podría ser la clasificación de valores. Para ello se usaremos la función analítica RANK(). La función de ventana RANK determina la clasificación de un valor en un grupo de valores, según la expresión ORDER BY definida en la cláusula OVER. Si dos registros tienen el mismo valor, la función RANK( ) asignará el mismo rango a ambos registros omitiendo el siguiente rango. Si hay una cláusula opcional PARTITION BY, las clasificaciones de restablecen para cada grupo de filas. La cláusula ORDER BY de una función analítica funciona independientemente de la cláusula ORDER BY de la query general que contiene la función analítica.

```
select
    employee_id,
    department_id,
    last_name,
    salary,
    RANK()
    OVER(PARTITION BY department_id
         ORDER BY
             employee_id ASC
    ) posicion
from
    Employees
```



Nota

La función RANK() no toma argumentos pero necesita los paréntesis vacíos.



Nota

Otra función de clasificación es DENSE_RANK(), la diferencia con RANK() es que aunque ambas identifican las filas con valores iguales con la misma clasificación, con DENSE:RANK no hay brecha de secuencia. Es decir, si dos filas tienen clasificación 1, la siguiente tendrá clasificación 2.

Mediante las funciones analíticas también podríamos realizar cálculos que nos permita mostrar cierta información de los datos:

```
select
  first_name,
  last_name,
  department_id,
  salary,
  ROUND(100 * salary / SUM(salary) OVER(PARTITION BY department_id), 2)
           AS porcentaje_del_total
from employees;
```

En este último ejemplo podemos observar el cálculo de un porcentaje basado en el total de los salarios de cada departamento. Se usa la función ROUND() para redondear el valor del promedio al entero más cercano. Por otro lado, dos de las funciones analíticas que introduce Oracle en su lenguaje SQL son LAG y LEAD, que permiten obtener el valor de la fila anterior y posterior respectivamente.

Esta funcionalidad es sumamente útil en operaciones de análisis de datos. Por ejemplo, realizaremos una consulta para saber cuál es el empleado más antiguo en cada departamento.

```
select
    employee_id,
    first_name,
    department_id,
    hire_date,
    LAG(hire_date)
    OVER(PARTITION BY department_id
         ORDER BY
             hire_date
    ) AS prev_hiredate
from
    employees;
```

FUNCION LISTAGG

Un requisito común de los informes comerciales es que los datos se muestren horizontalmente en vez de mostrarse verticalmente. Para entender esto haremos la siguiente consulta:

```
select
    department_id,
    last_name,
    first_name
from
    employees
order by
    department_id,
    last_name,
    first_name;
```

Esta consulta enumera todos los empleados junto con sus respectivos códigos de departamento, ordenados por departamento y nombre de empleado. Ahora probemos la siguiente consulta:

```
select
    department_id,
    LISTAGG(first_name
            || ' '
            || last_name, ', ') WITHIN GROUP
   (
    ORDER BY
        last_name, first_name
    ) employees
from
    employees
group by
    department_id
order by
    department_id;
```

En esta versión podemos ver los datos de la siguiente manera:

```
1        Jennifer Whalen
2        Pat Fay, Michael Hartstein
3        Shelli Baida, Karen Colmenares, Guy Himuro, Alexander Khoo, Den
          Raphaely, Sigal Tobias
4        Susan Mavris
5        Sarah Bell, Britney Everett, Adam Fripp, Payam Kaufling, Irene
          Mikkilineni, Shanta Vollman, Matthew Weiss
6        David Austin, Bruce Ernst, Dolores Garcia, Alexander Hunold, Diana
         Lorentz, Valli Pataballa
7        Hermann Baer
8        Kimberely Grant, Charles Johnson, Jack Livingston, Karen Partners,
         John Russell, Jonathon Taylor
9        Lex De Haan, Steven King, Neena Kochhar
10       John Chen, Daniel Faviet, Nancy Greenberg, Luis Popp, Ismael Sciarra,
          Jose Manuel Urman
11       William Gietz, Shelley Higgins
```

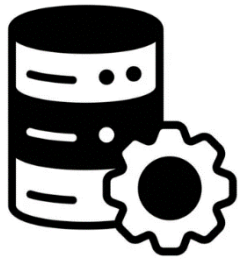
Para poder ver los datos de esta forma se usó la función LISTAGG. La función LISTAGG se utiliza, para pivotar filas en una única columna concatenando los valores con un separador indicado. Como se utilizó junto a esta función una cláusula GROUP BY se armó un grupo por cada código de departamento. La cláusula WITHIN GROUP (ORDER BY) especifica el orden de los valores agrupados.

VISTAS EN SQL

Esencialmente una vista (view) es un objeto con el resultado de una consulta SQL. Las vistas permiten a los usuarios acceder a los datos de manera más fácil y rápida, ya que no necesitan escribir consultas complejas. Las vistas se pueden usar también para reforzar la seguridad al limitar el acceso a datos confidenciales. Por ejemplo, puede crear una vista que solo muestre el nombre y apellido de los empleados, pero no su salario u otra información

confidencial del empleado. Crear una vista en SQL es relativamente sencillo: simplemente use la instrucción CREATE VIEW, seguida del nombre de la vista y la consulta que desea usar para crear la vista:

```
create view <nombre_vista>
as
select …
```



Nota

Una vista no contiene los datos reales; contiene solo la definición de la vista en el diccionario de datos. Por eso muchos desarrolladores lo llaman tablas virtuales.

Para este ejemplo construiremos una vista que permita devolver la antigüedad en años de cada empleado:

```
create or replace view employee_details as
select e.employee_id, e.first_name || ' ' || e.last_name AS full_name,
e.hire_date, TRUNC(MONTHS_BETWEEN(SYSDATE, e.hire_date) / 12) as
years_of_service, d.department_name, j.job_title, e.salary, m.first_name || '
' || m.last_name as manager_name, l.city || ', ' || l.country_id as
work_location
from employees e inner join departments d
on e.department_id = d.department_id inner join jobs j
on e.job_id = j.job_id inner join employees m
on e.manager_id = m.employee_id inner join locations l
on d.location_id = l.location_id order by e.employee_id;
```

Esta vista denominada employee_details reúne información de varias tablas (employees, departments, jobs, locations) para proporcionar un panorama completo de los detalles de cada empleado, incluyendo:

- Información básica del empleado (ID, nombre completo)
- Fecha de contratación y antigüedad en la empresa
- Departamento y puesto laboral
- Salario
- Nombre del superior
- Ubicación de trabajo (ciudad y país)

Si quisiéramos ver todas las vistas que encontraremos como usuarios debemos escribir lo siguiente:

```
select object_name, object_type  from user_objects  where object_type='VIEW'
order by object_type, object_name
```

También podemos hacer un describe para poder ver la estructura de nuestra vista:

```
describe employee_details
```

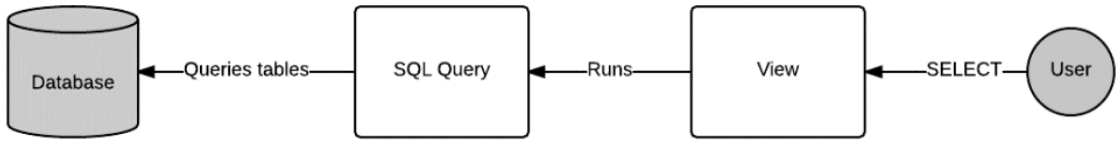
Para poder usar la vista recién creada debemos escribir lo siguiente:

```
select * from employee_details;
```

Como puede ver es mucho más simple ejecutar la vista que escribir una y otra vez la consulta compleja. También podemos realizar la siguiente consulta para probar nuestra vista:

```
select * from employee_details where full_name like 'B%'
```

En este caso devolverá toda la informacion de los empleados cuyo nombre completo comience con la letra B. En el diagrama que se muestra a continuación podemos ver el proceso de ejecución de una vista:



Podemos también ver el código de la vista usando la siguiente query:

```
select view_name, text from user_views;
```

Para poder eliminar una vista deberá escribir lo siguiente:

```
drop view view_name
```

Por ejemplo, para eliminar la vista recién creada debe tipear el siguiente comando:

```
drop view employee_details;
```

GraphQL EN ORACLE 23

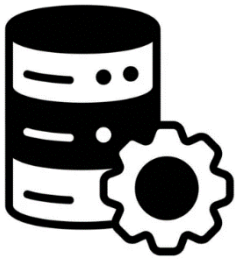
GraphQL es un lenguaje de consulta para APIs desarrollado por Facebook en 2012 como respuesta a los desafíos que enfrentaban sus productos durante el auge de los dispositivos móviles. Surgió cuando las limitaciones de las APIs RESTful se volvieron evidentes, particularmente en entornos con conexiones de red inestables o de baja velocidad, donde cada solicitud adicional aumentaba significativamente los tiempos de carga en los navegadores web móviles y el consumo de datos en los Smartphones. Las grandes cantidades de datos que se devolvían, combinadas con varias llamadas a la API, dieron como resultado una experiencia de usuario deficiente. Cuando utiliza una API RESTful para recuperar datos, a menudo tiene que realizar varias llamadas a la API para obtener toda la información que busca. Por ejemplo, si está iniciando sesión en su banco, primero debe obtener su número de cliente. Una vez que tenga su número de cliente, puede obtener su número de cuenta. Desde allí, puede obtener el saldo de su cuenta:

1. El login retorna el número de Cliente.
2. El numero de cliente retorna el numero de cuenta del cliente.
3. El numero de cliente retorna el saldo.

No hay nada de malo con las API REST, pero las múltiples llamadas API suelen ser lentas, con muchos datos e ineficientes. En el ejemplo anterior, para obtener el saldo de su cuenta bancaria, tuvo que crear una secuencia de tres llamadas API encadenadas, cada una de las cuales es una solicitud adecuada y una respuesta completa definida por el esquema. Una solicitud de GraphQL es más sencilla y específica, ya que trata solo los tres valores de ID de cliente, ID de cuenta y saldo:

```
query {
       customerId(name:”Joe Chesky”){
                accountId{
                    balance
                }
       }
}
```

El ejemplo anterior puede parecer sencillo, pero muestra claramente cómo GraphQL permite unificar en una sola consulta lo que tradicionalmente requeriría tres llamadas API independientes. La mejor parte es que no requiere la reimplementación de su funcionalidad existente.



Nota

Una sola consulta de GraphQL bien escrita puede devolver información que requeriría varias llamadas a la API RESTful con solicitudes HTTP. Los desarrolladores han descubierto que las llamadas de GraphQL devuelven conjuntos de datos que son un 90 % más pequeños que el uso de API RESTful para adquirir los mismos datos.

Puedo ejecutar GraphQL en Oracle 23ai usando una instrucción SELECT como en este caso:

```
select * from graphql('
emp : employees
{
  empno : employee_id
  name : first_name
  job : job_id
  sal : salary
  hiredate : hire_date,
  depto: departments
  {
   depto_id:department_id,
   depto_name:department_name
  }
}');
```

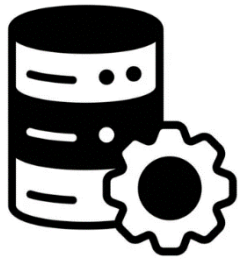
Como podemos observar, la salida de la instrucción SELECT es en formato JSON. En este caso devolvemos la información de los empleados con datos de sus departamentos. Quizás no queramos todos los empleados sino aquellos que tengan un job_id como IT_PROG, para realizar este filtro vamos a usar la siguiente instrucción SELECT:

```
select * from graphql('
emp : employees (where : {job_id: {_eq : "IT_PROG"}})
{
  empno : employee_id
  name : first_name
  job : job_id
  sal : salary
  hiredate : hire_date,
  depto: departments
  {
   depto_id:department_id,
   depto_name:department_name
  }
}');
```

VISTAS DE DUALIDAD RELACIONAL EN ORACLE

La convergencia entre SQL y NoSQL representa una evolución significativa en el panorama tecnológico actual, permitiendo a los desarrolladores beneficiarse de la simplicidad del desarrollo de documentos JSON y la potencia de las bases de datos relacionales con una única base de datos. Las bases de datos multimodelo permiten una solución superando las limitaciones del modelo relacional y NoSQL y también beneficiándose de ambos modelos. Esta evolución elimina el tradicional dilema de elegir entre paradigmas, permitiendo a los equipos de desarrollo utilizar el enfoque más adecuado según los requerimientos específicos de las aplicaciones., disfrutan de la flexibilidad y la libertad de elegir el mejor enfoque para cada aplicación sin sacrificar la funcionalidad. Oracle 23ai incorpora las vistas de dualidad relacional-JSON (JSON Relational Duality Views). Las vistas de dualidad JSON permite mapear tablas relacionales a JSON dinámicamente. Esto permite que la base de datos almacene datos relacionalmente, pero los exponga como JSON sin duplicarlos. De esta forma admite la manipulación bidireccional de datos: las actualizaciones JSON se traducen en actualizaciones relacionales y viceversa. En resumen:

- Las vistas de dualidad proporcionan una interfaz JSON sobre datos relacionales.
- Las lecturas y escrituras se realizan como JSON, pero Oracle las almacena en tablas relacionales.
- Admite la manipulación de datos bidireccional: las actualizaciones JSON se traducen en actualizaciones relacionales y viceversa.



Nota

Las Vista de Dualidad Relacional, permite almacenar datos en tablas relacionales normalizadas, pero se accede a ellos como documentos JSON. Este enfoque combina lo mejor de ambos mundos, ofreciendo a los desarrolladores la simplicidad de JSON para el acceso a los datos y la robustez del modelo relacional para el almacenamiento.

Vamos a crear una vista de dualidad JSON usando graphQL:

```
create or replace json relational duality view department_dv as
departments @insert @update @delete{
    _id : department_id,
    departmentName : department_name,
    location : location_id,
    employees : employees @insert @update @delete{
        employeeNumber : employee_id,
        employeeFirst : first_name,
        employeeLast : last_name,
        employeeEmail: email,
        employeeDate: hire_date,
        job : job_id,
        salary : salary
    }
};
```

Usando la potencia de graphQL hemos creado una vista que nos proporciona todos los empleados que trabajan en cada departamento de la organización en formato JSON. Observe el uso de las anotaciones @insert, @update y @delete que proporcionar un nivel adecuado de acceso de escritura a las tablas de la vista.

Para poder ver el resultado haremos la siguiente consulta:

```
select * from department_dv;
```

Podemos consultar datos de SQL utilizando la notación de puntos. Si quisiera averiguar la información que tengo del departamento IT debe realizar la siguiente consulta:

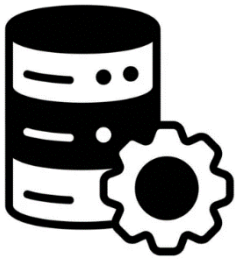
```
select * from department_dv where json_value(data, '$.departmentName') = 'IT'
```

Si quisiéramos insertar un nuevo empleado en un nuevo departamento deberíamos realizar la siguiente instrucción INSERT:

```
insert into department_dv (data)
values ('
{
  "_id" : 300,
  "departmentName" : "Legal",
  "location" : 2400,
  "employees" : [
    {
      "employeeNumber" : 9999,
      "employeeFirst" : "Emma",
      "employeeLast" : "Roberts",
      "employeeEmail" : "EROBERTS@mycompany.com",
      "employeeDate": "2024-06-07T00:00:00",
      "job" : "PR_REP",
      "salary" : 8000
    }
  ]
}');
```

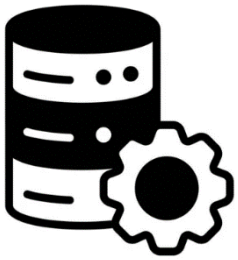
ORACLE 23ai Y LA INTELIGENCIA ARTIFICIAL

En el campo de la inteligencia artificial (IA), y el machine learning grandes cantidades de datos requieren una gestión y procesamiento eficientes. A medida que nos adentramos en aplicaciones más avanzadas de la IA, como el reconocimiento de imágenes, la búsqueda por voz o los sistemas de recomendaciones, la naturaleza de los datos se vuelve más complejos. Aquí es donde entran en juego las bases de datos vectoriales. A diferencia de las bases de datos tradicionales, que almacenan valores escalares, las bases de datos vectoriales están especialmente diseñadas para gestionar puntos de datos multidimensionales, a menudo denominados vectores. A partir de la versión 23ai, Oracle introduce el tipo de dato vector. Un vector es una estructura de datos popular que se utiliza en aplicaciones de IA. Básicamente es una lista de números, generada por modelos de aprendizaje profundo a partir de diversos tipos de datos como imágenes, audios, videos, entradas de blogs, etc., esta lista representa la semántica de dichos valores. Gracias al uso de vectores no hace falta almacenar todo el archivo dentro de la base de datos simplemente se transforma dicha información en una lista de números. La ventaja de este tipo de almacenamiento es que se pueden realizar búsquedas por similitud (Vector Search) más allá de las búsquedas tradicionales basadas en palabras claves.



Nota

La búsqueda vectorial (Vector Search) es una potente tecnología que realiza búsquedas basadas en la similitud o distancia de datos representados como vectores multidimensionales. A menudo se considera más eficaz que la búsqueda basada en palabras clave, ya que se basa en el significado y en el contexto de las palabras, y no en las palabras en sí. Gracias a esto un jefe de RRHH podría encontrar al mejor candidato para un puesto laboral.



Nota

Una base de datos vectorial es cualquier base de datos que pueda almacenar y administrar de forma nativa incrustaciones vectoriales y manejar los datos no estructurados que describen, como documentos, imágenes, videos o audio.