Complementos de SQL

Alexandre Pereiras Magariños
M. Elena Rodríguez González

PID_00235094

Universitat Oberta de Catalunya

Índice

Introducción..... 5

Objetivos..... 6

1. Claves subrogadas..... 7
1.1. Concepto 7
1.2. Beneficios de las claves subrogadas en el diseño de almacenes de datos 10
1.3. Construcción de claves subrogadas en PostgreSQL 12
1.3.1. Secuencias 13
1.3.2. Tipo de dato *serial*..... 13

2. *Common table expression*..... 15
2.1. Concepto 15
2.2. Beneficios del uso de CTE 18
2.3. Construcción de CTE en PostgreSQL 18
2.3.1. Consultas recursivas 19
2.3.2. Consultas CTE con sentencias de manipulación de datos 27

3. Funciones analíticas..... 29
3.1. Concepto 29
3.2. Beneficios de las funciones analíticas 35
3.3. Funciones analíticas en PostgreSQL 35
3.4. Tipos de funciones analíticas en PostgreSQL 40
3.4.1. *Row number*..... 41
3.4.2. *Rank*..... 43
3.4.3. *Dense rank*..... 44
3.4.4. *Lag*..... 45
3.4.5. *Lead*..... 46
3.4.6. *First value*..... 48
3.4.7. *Last value*..... 49
3.5. Uso de funciones de agregación como funciones analíticas 51

4. Tratamiento de valores nulos..... 54
4.1. Valores nulos en BD operacionales 54
4.2. Valores nulos en almacenes de datos 56
4.2.1. Valores nulos en tablas de dimensiones 57
4.2.2. Valores nulos en tablas de hechos 62

5. Transacciones..... 65

5.1. Problemática asociada a la gestión de transacciones 65
5.2. Definición y propiedades de las transacciones 68
5.3. Interferencias entre transacciones 69
5.4. Nivel de concurrencia 75
5.5. Visión externa de las transacciones 76
5.5.1. Relajación del nivel de aislamiento 80
5.5.2. Responsabilidades del SGBD y del desarrollador 81
5.6. Transacciones en PostgreSQL 83
5.7. Importancia de las transacciones en OLTP frente a OLAP 84

Resumen..... 87

Ejercicios de autoevaluación..... 89

Solucionario..... 92

Glosario..... 97

Bibliografía..... 99

Anexos: complementos de SQL – anotaciones para Oracle..... 100

Introducción

En este módulo didáctico ampliaremos los conocimientos que tenemos del SQL estándar; más concretamente, estudiaremos un conjunto de funcionalidades que los sistemas de gestión de bases de datos (SGBD) implementan y que son muy útiles para el desarrollo de aplicaciones en entornos *data warehouse*. Concretamente, trabajaremos con los conceptos de claves subrogadas, *common table expression* y funciones analíticas, conceptos que resultan de gran utilidad a la hora de implementar procesos de carga de datos (ETL) y desarrollo de informes analíticos.

BD

Base de datos

Además, como información complementaria, estudiaremos la problemática de la existencia de valores nulos tanto en bases de datos (BD) operacionales como en almacenes de datos, y cómo solventar de manera eficiente estos problemas.

SGBD

Sistema gestor de bases de datos

Por último, veremos el concepto de transacción y sus propiedades, y estudiaremos la problemática del acceso simultáneo a los datos por parte de los usuarios, y cómo los SGBD gestionan estos escenarios de manera segura.

Hay que tener en cuenta que a menudo existen diferencias entre lo que dice el estándar SQL (cuya última versión en el momento de escribir este módulo es SQL:2012) y las implementaciones de los diversos proveedores de SGBD relacionales. En cada una de las secciones de este módulo se muestra, a modo de ejemplo, cómo aplicar los conceptos explicados utilizando el SGBD relacional PostgreSQL. Como información complementaria, se proporciona un anexo en el que se describe la implementación de estos mismos conceptos utilizando en este caso el SGBD relacional Oracle.

ETL

Del inglés *extract, transform and load*, son el conjunto de procesos en entornos *data warehouse* que se encargan de la extracción de datos procedentes de múltiples orígenes, de la transformación de estos para adecuarlos a las nuevas estructuras, y de su carga final en el almacén de datos para su consumo.

Las versiones utilizadas de estos dos SGBD relacionales son PostgreSQL 9.3 y Oracle 11g respectivamente, por lo que en sucesivas versiones será necesario consultar los manuales de cada uno de los fabricantes para identificar, si es el caso, los cambios respecto a la documentación presentada en este módulo didáctico.

Almacén de datos

En inglés *data warehouse*, son bases de datos orientadas a áreas de interés de la empresa que integran datos de distintas fuentes con información histórica y no volátil, y que tienen como objetivo principal servir de apoyo en la toma de decisiones.

Objetivos

Este módulo didáctico presenta conceptos avanzados de SQL para el tratamiento de datos. Una vez finalizado su estudio, tendréis las herramientas necesarias para alcanzar los siguientes objetivos:

1. Entender el concepto de clave subrogada y los diferentes mecanismos utilizados para su generación.

2. Conocer, entender y generar consultas *common table expression*.

3. Conocer, entender y generar consultas SQL recursivas.

4. Conocer, de forma general, las características y funcionalidad de las funciones analíticas, y saber aplicarlas para la obtención de cálculos complejos.

5. Entender la problemática de los valores nulos en BD operacionales y entornos *data warehouse*, y aplicar las soluciones apropiadas.

6. Comprender qué es una transacción, sus propiedades, y las funciones que tiene que cumplir un SGBD en la gestión de transacciones.

7. Ser capaz de desarrollar aplicaciones que utilicen de forma correcta y eficiente los servicios de gestión de transacciones que ofrecen los SGBD.

1. Claves subrogadas

En los SGBD, las claves primarias de una tabla pueden definirse a partir de una sola columna, denominándose **clave primaria simple**, o bien como una combinación de columnas, llamándose esta última **clave primaria compuesta**. Existen casos en los que el SGBD incurre en un cierto nivel de ineficiencia a la hora de realizar operaciones de combinación en consultas, ineficiencia que se observa cuando el diseñador se ve obligado a generar las claves primarias de las tablas a partir de una cantidad considerable de columnas (lo que requiere que el SGBD realice operaciones de combinación sobre múltiples columnas), o incluso aun siendo una clave primaria simple, cuando los valores de dicha columna son lo suficientemente grandes en tamaño (asumiendo en este último caso un tipo de datos alfanumérico). Esta ineficiencia podría reducirse o eliminarse si el tipo de dato de la clave primaria fuese numérico y de menor tamaño, y por consiguiente, fuese una clave primaria simple. Con el fin de mejorar esta problemática, se propone el concepto de **claves subrogadas**, concepto que veremos en las siguientes secciones.

1.1. Concepto

El concepto de claves subrogadas es muy utilizado dentro de implementaciones de *data warehouse* y en entornos de *business intelligence*. Para entender el concepto de **clave subrogada**, es necesario definir primero el concepto de **clave de negocio**.

Modelos multidimensionales

Un modelo multidimensional es aquel que concibe los datos que queremos analizar en términos de hechos y dimensiones de análisis, de modo que los podemos situar en un espacio n-dimensional.

Entendemos por **clave de negocio** (en inglés, *business key*) el conjunto de columnas que conforman la clave primaria de una tabla, que generalmente tiene un significado propio acorde con las reglas de negocio del sistema.

Claves subrogadas

Las claves subrogadas también reciben el nombre de claves sin significado (*meaningless keys*), claves enteras (*integer keys*), claves artificiales (*artificial keys*), claves no naturales (*non-natural keys*) o claves sustitutas (*substitute keys*).

Se entiende por **clave subrogada** (en inglés, *surrogate key*) el identificador único de una tabla que no se deriva de los datos de la aplicación y que generalmente no es visible al usuario. Suele construirse a partir de una secuencia numérica autogenerada (con valores enteros, sin decimales) en el que no existe una relación entre el significado de la fila y dicha clave. Una clave subrogada está siempre conformada por una única columna.

La distinción entre clave de negocio y clave subrogada está muy presente en **modelos multidimensionales** y los procesos ETL. Dentro de este contexto, la **clave subrogada** representa la **clave primaria de la tabla destino**, y **la clave de negocio** representa la **clave primaria de la tabla origen** (desde la que se leen los datos) y que también se suele almacenar en la tabla destino.

Ejemplo de claves de negocio y claves subrogadas en un modelo multidimensional

Supongamos que tenemos la tabla Asignatura con cuatro columnas: id, código, nombre de la asignatura y fecha de alta. En este ejemplo, id es clave primaria y clave subrogada (no tiene un significado aparente, es una secuencia numérica), y código es la clave de negocio (con un significado especial dentro del sistema origen).

Asignatura			
Id	Código	Nombre	Fecha Alta
1	UOC-12345	Bases de datos	10/01/2000
2	UOC-66521	Ingeniería del software	10/01/2000
3	UOC-98321	Programación	10/01/2000
4	UOC-21473	Álgebra	10/01/2000

El hecho de mantener esta separación en modelos multidimensionales facilita el control de los cambios que los sistemas origen puedan sufrir en el caso de que estos reutilicen los valores de la clave de negocio a lo largo del tiempo. De esta manera, preservamos el histórico de cambios para facilitar el análisis de los datos.

Ejemplo de reutilización de claves de negocio

Es frecuente que las BD operacionales reutilicen valores de clave primaria tras un período de inactividad. Un caso concreto sería el de los números de teléfono móvil.

Supongamos que tenemos una base de datos con una tabla de números de teléfono de una compañía de telecomunicaciones, en la que se guarda el número de teléfono y el nombre del titular. La clave primaria de esta tabla es el número de teléfono.

Números de Teléfono	
Número Teléfono	Nombre Titular
655138007	Manuela Domínguez
655138006	José López
655138005	Juan Manuel Carrillo

En el caso en el que un cliente da de baja un número de teléfono, el número podría ser asignado de nuevo a otro cliente tras un período de inactividad (por ejemplo, doce meses). Este sería el caso del número 655138007, que, como se puede ver en la siguiente tabla, ha cambiado el titular a José Sánchez, a diferencia del titular que previamente tenía asignado dicho número de teléfono (Manuela Domínguez).

Base de datos operacional

Base de datos destinada a gestionar el día a día de una organización, es decir, almacena la información en lo referente a la operativa diaria de una institución.

Números de Teléfono	
Número Teléfono	**Nombre Titular**
655138007	José Sánchez
655138006	José López
655138005	Juan Manuel Carrillo

La problemática es que, en este escenario, no mantenemos un histórico de los titulares. Este escenario se puede representar muy bien en modelos multidimensionales.

Véase ahora el siguiente ejemplo, donde tenemos una dimensión de números de teléfono (que utiliza la tabla de la base de datos operacional como origen de datos) con una clave subrogada como clave primaria. Vemos que el número de teléfono 655138007 puede reutilizarse y mantener el titular original, separando así la clave de negocio de la clave subrogada. De esta forma podríamos, por ejemplo, asignar las llamadas realizadas por cada cliente de forma correcta.

Dimensión Números de Teléfono		
Id	**Número Teléfono**	**Nombre Titular**
1	655138007	Manuela Domínguez
2	655138006	José López
3	655138005	Juan Manuel Carrillo
4	655138007	José Sánchez

Almacén de datos

Los almacenes de datos (en inglés, *data warehouse*) son BD orientadas a áreas de interés de la empresa que integran datos de distintas fuentes con información histórica y no volátil, y que tienen como objetivo principal servir de apoyo en la toma de decisiones.

Existe un caso especial en el que la clave subrogada podría tener un significado especial. A la hora de diseñar un almacén de datos, existe una tabla muy importante denominada **Fecha** o **Día**, que representa los días del calendario, es decir, cada fila almacenará una fecha concreta: 1 Enero 2015, 15 Julio 2002, etc. y las características asociadas a cada fecha (día de la semana, si es fin de semana, si es festivo...). En el caso especial de esta tabla, la clave subrogada se suele representar como un entero cuyo valor es la fecha representada en formato YYYYMMDD. Utilizando los ejemplos de fechas anteriores, el 1 Enero 2015 se representaría como 20150101, y el 15 Julio 2002 se representaría como 20020715.

Ejemplo de dimensión Fecha

Véase el ejemplo propuesto para una dimensión Fecha, donde la clave primaria (y subrogada) es la fecha en formato YYYYMMDD.

Dimensión Fecha						
Id Fecha	**Fecha**	**Día de la Semana**	**Mes**	**Mes (Dígitos)**	**Trimestre**	**Año**
20150101	01/01/2015	Jueves	Enero	1	Q1	2015
20150102	02/01/2015	Viernes	Enero	1	Q1	2015
20150103	03/01/2015	Sábado	Enero	1	Q1	2015

Dimensión Fecha						
Id Fecha	**Fecha**	**Día de la Semana**	**Mes**	**Mes (Dígitos)**	**Trimestre**	**Año**
20150104	04/01/2015	Domingo	Enero	1	Q1	2015

Una de las razones principales por la que la clave subrogada de la dimensión Fecha tiene un significado es, a diferencia de otras, para facilitar un particionamiento físico de los datos eficiente: la posibilidad de separar datos de forma física en base al valor de la columna Fecha permite no solamente mejorar el rendimiento de la consulta a la hora de consultar los datos de forma histórica, sino su mantenimiento, facilitando así la inserción de nuevos datos y el purgado de datos históricos sin que estas dos operaciones se afecten mutuamente.

1.2. Beneficios de las claves subrogadas en el diseño de almacenes de datos

El uso de claves subrogadas en el ámbito de los almacenes de datos, además de ser una buena práctica a la hora de diseñarlos, lleva asociado una serie de beneficios.

Como ya se ha mencionado anteriormente, una de las principales ventajas de utilizar claves subrogadas es la de crear una separación en modelos multidimensionales para facilitar el control de los cambios en los sistemas origen a la hora de reutilizar valores de la clave de negocio (véase el ejemplo de los números de teléfono mostrado anteriormente).

Otra ventaja muy importante es la mejora de rendimiento en operaciones de consulta. Las operaciones de combinación (`JOIN`) entre las diferentes tablas dentro de un modelo multidimensional (dimensiones y hechos) se realizarán mediante estas claves subrogadas, cuyo tipo de datos es, como ya hemos mencionado, numérico-entero, y que comparados con otros tipos de datos como fechas o cadenas de caracteres alfanuméricas, suelen ocupar menos espacio. El hecho de que ocupen menos espacio significa que las tablas de hechos, que contienen las claves foráneas de las dimensiones, sean más pequeñas, y por lo tanto sus índices serán también más pequeños. Esto se traduce en una reducción en el número de páginas utilizadas para almacenar filas (podemos almacenar más filas por página), por lo que podremos leer más datos con menos operaciones de entrada/salida (E/S).

Otro de los beneficios importantes del uso de claves subrogadas es el de facilitar el **seguimiento de cambios surgidos en las BD operacionales** para facilitar el análisis de datos y mantener el histórico de valores y descripciones (al contrario que en un sistema operacional, tal y como hemos visto en el ejemplo de reutilización de números de teléfono).

Referencia bibliográfica

Para ver las diferentes técnicas de seguimiento de cambios en un almacén de datos, se recomienda revisar la siguiente referencia bibliográfica:

Kimball, R.; Ross, M. (2013). *The Data Warehouse Toolkit* (3.ª ed.). John Wiley & Sons, Inc.

Ejemplo de seguimiento de cambios en un modelo multidimensional

Supongamos que, en el ejemplo de la tabla Asignatura, la asignatura con código UOC-21473 (Álgebra) ya no se imparte más y su código es reutilizado por la asignatura Programación Orientada a Objetos. En este escenario, se añadiría una nueva fila con el mismo código, y el nuevo nombre y fecha de alta.

Asignatura			
Id	**Código**	**Nombre**	**Fecha Alta**
1	UOC-12345	Bases de datos	10/01/2000
2	UOC-66521	Ingeniería del software	10/01/2000
3	UOC-98321	Programación	10/01/2000
4	UOC-21473	Álgebra	10/01/2000
5	UOC-21473	Programación Orientada a Objetos	15/09/2015

Las claves subrogadas facilitan también la **integración y consolidación de datos desde múltiples orígenes de datos**, aun cuando la **clave de negocio** en los diferentes sistemas **no sea homogénea** en todos ellos.

Ejemplo de uso de claves subrogadas para la integración de datos entre múltiples orígenes

Supongamos que tenemos dos sistemas que almacenan información de usuario. Los usuarios en cada sistema se codifican de manera diferente: el primer sistema utiliza nombres de usuario en formato UXXXXX, donde XXXXX es una secuencia numérica, mientras que el segundo sistema utiliza un formato ADDXXXX, donde DD es el código de departamento y XXXX es un secuencia numérica. Este escenario se podría representar en un modelo multidimensional con claves subrogadas de la siguiente manera:

Usuario		
Id	**Id Usuario**	**Sistema Origen**
1	U00001	SISTEMA 1
2	U00002	SISTEMA 1
3	AHR0001	SISTEMA 2
4	AHR0002	SISTEMA 2
5	U00003	SISTEMA 1

El uso de claves subrogadas también nos permite codificar de forma eficiente la **falta de valores en una base de datos operacional**. De esta forma, podemos asignar un valor de clave subrogada a un valor **por defecto** y mapearlo correctamente a la tabla de hechos.

Ejemplo de valores por defecto

Es común que exista una fila con una clave subrogada especial para mapear la inexistencia de valores en la base de datos operacional. En el ejemplo de Asignatura, se ha añadido una fila con clave subrogada -1 que se utilizará para identificar la falta de códigos de asignaturas:

Asignatura		
Id	**Código**	**Nombre**
-1	Desconocida	Desconocida
1	UOC-12345	Bases de datos
2	UOC-66521	Ingeniería del software
3	UOC-98321	Programación
4	UOC-21473	Álgebra

1.3. Construcción de claves subrogadas en PostgreSQL

En esta sección vamos a trabajar cómo se pueden construir claves subrogadas utilizando el SGBD PostgreSQL. Para ello, supongamos que disponemos de la siguiente definición de la tabla Asignatura: *asignatura_key* (clave subrogada y clave primaria), *cod_asignatura* (código de asignatura) y *nom_asignatura* (nombre de la asignatura). A continuación podemos ver la definición de esta tabla en lenguaje SQL.

```
CREATE TABLE asignatura (
    asignatura_key  INTEGER PRIMARY KEY,
    cod_asignatura  CHARACTER VARYING(10)  NOT NULL,
    nom_asignatura  CHARACTER VARYING(100) NOT NULL
)
```

Notación

La notación para representar la sintaxis de las sentencias SQL será la siguiente:

- Las palabras en mayúsculas son palabras reservadas del lenguaje.
- Las palabras en minúsculas son nombres de estructuras de la BD creadas por el usuario (tablas, columnas, etc.).
- La notación [...] quiere decir que lo que hay entre los corchetes es opcional.
- La notación {A|...|B} quiere decir que hemos de escoger entre todas las opciones que hay entre las llaves, pero que debemos poner una obligatoriamente.

La generación de claves subrogadas se puede implementar de las siguientes formas: mediante el uso de secuencias y mediante el uso del tipo `serial`.

1.3.1. Secuencias

La creación de una secuencia nos permite generar valores numéricos de forma consecutiva. La definición de secuencias en PostgreSQL es la siguiente:

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
```

Como ejemplo, a continuación definiremos la secuencia que utilizará la tabla Asignatura:

```
CREATE SEQUENCE seq_asignatura_key INCREMENT BY 1 START WITH 1 NO CYCLE;
```

Para hacer uso de la secuencia y así generar nuevos valores, disponemos de la función `nextval(seq_name)`, donde `seq_name` representa el nombre de la secuencia a utilizar. Esta función se encarga de avanzar al siguiente valor de la secuencia y devolver dicho valor. Estas dos operaciones se realizan de forma atómica, por lo que múltiples sesiones pueden hacer uso de la misma secuencia sin preocuparse de que los valores se reutilicen.

Nota

La explicación de las opciones de las diferentes cláusulas utilizadas en la creación de una secuencia se puede consultar en la siguiente URL de PostgreSQL:
http://www.postgresql.org/docs/9.3/static/sql-createsequence.html

```
SELECT nextval('seq_asignatura_key')
```

A la hora de insertar una fila nueva en la tabla, podemos hacer una llamada a la función como parte de la sentencia `INSERT`:

```
INSERT
INTO
  asignatura
  (
    asignatura_key,
    cod_asignatura,
    nom_asignatura
  )
  VALUES
  (
    nextval('seq_asignatura_key'),
    'UOC-35298',
    'Arquitectura de Datos'
  )
```

Ejercicio

Investigar el uso de las funciones de manipulación de secuencias en la siguiente URL:
http://www.postgresql.org/docs/9.3/static/functions-sequence.html

1.3.2. Tipo de dato *serial*

Otra forma de definir claves subrogadas es mediante la creación de una columna con tipo de datos `serial` (o bien `smallserial` y `bigserial`, dependiendo de la capacidad numérica que se necesite para almacenar los valores).

Si decidimos utilizar este mecanismo, la tabla Asignatura se tendrá que definir de la siguiente manera (fijaos cómo la definición de la columna *asignatura_key* ha cambiado con respecto a la definición original):

```
CREATE TABLE asignatura (
    asignatura_key  SERIAL PRIMARY KEY,
    cod_asignatura  CHARACTER VARYING(10)  NOT NULL,
    nom_asignatura  CHARACTER VARYING(100) NOT NULL
)
```

Cada vez que se inserta una nueva fila en la tabla, la columna *asignatura_key* generará un nuevo valor. Véanse los siguientes ejemplos, en donde se pueden ver que se puede especificar el valor con la cláusula `DEFAULT` o, simplemente, no incluir dicha columna como parte de la sentencia `INSERT`:

```
INSERT INTO asignatura (asignatura_key, cod_asignatura, nom_asignatura)
VALUES (DEFAULT, 'UOC-35299', 'Sistemas Operativos');

INSERT INTO asignatura (cod_asignatura, nom_asignatura)
VALUES ('UOC-35298', 'Arquitectura de Datos');
```

En realidad, los tipos de datos `serial` y sus variantes no son tipos de datos reales. Estos no son más que una notación especial en PostgreSQL que se traduce en la creación de una secuencia y la asignación de un valor por defecto a la columna en cuestión. Utilizando el ejemplo anterior, el resultado de crear dicha tabla con este tipo de dato sería equivalente a especificar las siguientes estructuras:

```
CREATE SEQUENCE seq_asignatura_key;

CREATE TABLE asignatura (
    asignatura_key  INTEGER NOT NULL DEFAULT nextval('seq_asignatura_key') PRIMARY KEY,
    cod_asignatura  CHARACTER VARYING(10)  NOT NULL,
    nom_asignatura  CHARACTER VARYING(100) NOT NULL
);

ALTER SEQUENCE seq_asignatura_key
    OWNED BY asignatura.asignatura_key;
```

2. *Common table expression*

Cuando generamos consultas SQL, existen situaciones en las que necesitamos realizar operaciones o cálculos sobre un conjunto de datos que no existen en el sistema de forma inherente, sino que estos han de ser obtenidos mediante agregaciones, combinaciones entre tablas, filtros y cálculos sobre los datos existentes.

Una forma de obtener este conjunto de datos es mediante el uso de vistas, con las que conseguimos dividir aquellas consultas complejas en partes más sencillas, encapsulando así el código que necesitamos para obtener el subconjunto de datos de las tablas maestras que necesitamos. El problema de las vistas es que son objetos permanentes en el sistema, lo que puede resultar un inconveniente a la hora de crear consultas no planificadas. Por ejemplo, en un entorno de producción (entornos que suelen ser estrictos y restringidos a la hora de crear objetos en la BD). Otra solución que los SGBD nos proporcionan es la de generar subconsultas. Sin embargo, estas tienen el problema de que dificultan la lectura del código y su mantenimiento, y nos obliga, en ciertas situaciones, a repetir el mismo código en más de una ocasión, debido a la imposibilidad de referenciar una subconsulta a lo largo de la consulta principal. Para solventar esta problemática, se han introducido las *common table expression*, que nos ofrecen una forma más sencilla y elegante de generar consultas que requieren de datos no inherentes en el sistema, y que estos puedan referenciarse de manera sencilla.

A continuación, veremos con más detalle qué son las *common table expression* y cuáles son los beneficios de este tipo de construcciones, además de proporcionar las cláusulas en PostgreSQL necesarias para la generación de este tipo de consultas, cláusulas que han sido añadidas como parte del estándar SQL:1999.

2.1. Concepto

Las **CTE** (del inglés *common table expression*) son una funcionalidad que proporciona SQL para simplificar y facilitar la construcción de consultas complejas. Las CTE se crean a partir de la cláusula `WITH`.

Las CTE, mediante el uso de la cláusula `WITH`, nos permiten definir consultas auxiliares para su uso en consultas más complejas mediante una única declaración. Estas consultas auxiliares permiten «romper» dicha consulta compleja en consultas más pequeñas y legibles, además de permitir la reutilización de estas pequeñas consultas en más de una ocasión dentro de una misma consulta. Podríamos decir que estas consultas auxiliares tienen un comportamiento

similar a la construcción de tablas temporales, ya que se trata de consultas cuyos datos resultantes, de alguna manera, se guardan de forma temporal por el SGBD y se descartan una vez que la consulta ha finalizado su ejecución.

Ejemplo de consulta CTE

Supongamos que tenemos la siguiente tabla de empleados. La columna Id Supervisor es el identificador de empleado que actúa como supervisor. En el caso de que el empleado no tenga un supervisor asignado, esto significa que el empleado es el director general de la empresa.

Empleados				
Id Empleado	**Nombre**	**Ciudad**	**Salario Anual**	**Id Supervisor**
1	Manuel Vázquez	Barcelona	23500	7
2	Elena Rodríguez	Tarragona	16000	1
3	José Pérez	Girona	17000	1
4	Alejandra Martínez	Barcelona	22500	7
5	Marina Rodríguez	Vilanova	12000	4
6	Fernando Nadal	Viladecans	13000	4
7	Victoria Suarez	Tarragona	31000	10
8	Víctor Anllada	Lleida	28000	10
9	José María Llopis	Barcelona	29000	10
10	Victoria Setan	Castelldefels	45000	
11	Manuel Bertrán	Barcelona	21000	9

Dado este conjunto de datos, queremos obtener un listado de todos los empleados (nombre y ciudad), la diferencia entre el salario máximo y el salario del empleado, la diferencia entre el salario mínimo y el salario del empleado, y la diferencia entre el salario medio y el salario del empleado, ordenado por el identificador de empleado ascendentemente. El cálculo de los salarios máximo, mínimo y medio debe realizarse excluyendo al director general. Esta consulta se podría implementar de la siguiente manera:

```
SELECT
  e1.nombre,
  e1.ciudad,
  e1.salario - (SELECT MAX(e2.salario)
                  FROM empleado e2
                 WHERE e2.id_supervisor IS NOT NULL) AS diferencia_max,
  e1.salario - (SELECT MIN(e3.salario)
                  FROM empleado e3
                 WHERE e3.id_supervisor IS NOT NULL) AS diferencia_min,
  e1.salario - (SELECT AVG(e4.salario)
                  FROM empleado e4
                 WHERE e4.id_supervisor IS NOT NULL) AS diferencia_avg
FROM
  empleado e1
ORDER BY
  e1.id_empleado
```

Como podemos ver, el código para obtener los salarios máximo, mínimo y medio es muy similar: las tres subconsultas utilizan la misma tabla y la misma condición en la cláusula `WHERE`. En cambio, la consulta repite el mismo código varias veces (tabla y condiciones impuestas) en diferentes partes. ¿Qué pasaría si ahora tuviésemos que eliminar la condición de no incluir al director general? ¿Qué pasaría si en lugar de excluir al director ge-

neral debiésemos excluir también a los empleados de Barcelona? En estos casos, tendríamos que modificar el código hasta en tres sitios diferentes, lo que sería en cierto modo ineficiente y difícil de mantener.

Utilizando una consulta CTE construiríamos la consulta de la siguiente manera. Ved que toda la lógica de la consulta está escrita en un lugar en concreto mediante la cláusula `WITH` (como si fuese una tabla temporal denominada `salarios`), y que esta es luego referenciada para calcular las diferencias. En el caso de que necesitemos modificar los criterios de selección de los salarios, bastaría con modificar la definición de la consulta auxiliar llamada `salarios`. De esta forma, se simplifica el código y se facilita tanto la lectura de este como su mantenimiento, además de mejorar el rendimiento de la consulta debido a que `salarios` se evalúa una única vez.

```
WITH salarios AS (
    SELECT
        MAX(salario) AS max_salario,
        MIN(salario) AS min_salario,
        AVG(salario) AS avg_salario
    FROM
        empleado
    WHERE
        id_supervisor IS NOT NULL
)
SELECT
    nombre,
    ciudad,
    salario - (SELECT max_salario FROM salarios) AS diferencia_max,
    salario - (SELECT min_salario FROM salarios) AS diferencia_min,
    salario - (SELECT avg_salario FROM salarios) AS diferencia_avg
FROM
    empleado
ORDER BY
    id_empleado
```

Los resultados de ambas consultas, que son equivalentes, se muestran a continuación:

Nombre	Ciudad	Diferencia Máx	Diferencia Mín	Diferencia Avg
Manuel Vázquez	Barcelona	-7500	11500	2200
Elena Rodríguez	Tarragona	-15000	4000	-5300
José Pérez	Girona	-14000	5000	-4300
Alejandra Martínez	Barcelona	-8500	10500	1200
Marina Rodríguez	Vilanova	-19000	0	-9300
Fernando Nadal	Viladecans	-18000	1000	-8300
Victoria Suarez	Tarragona	0	19000	9700
Víctor Anllada	Lleida	-3000	16000	6700
José María Llopis	Barcelona	-2000	17000	7700
Victoria Setan	Castelldefels	14000	33000	23700
Manuel Bertrán	Barcelona	-10000	9000	-300

2.2. Beneficios del uso de CTE

De los posibles beneficios del uso de *common table expression*, podemos destacar los siguientes:

CTE Bounties

En la wiki de PostgreSQL CTE Bounties se presentan varios ejemplos de qué tipo de problemas podemos resolver con el uso de CTE y la solución propuesta. El enlace a dicho wiki es el siguiente:
https://wiki.postgresql.org/wiki/CTE_Bounties

1) Facilitar la legibilidad y mantenimiento del código: esta es una característica muy importante, ya que nos permite definir cálculos complejos una única vez, y ser reutilizados en diferentes puntos de una misma consulta.

2) Generar y reutilizar código de forma más eficiente: una de las propiedades de las CTE es que se evalúan una sola vez por ejecución, por lo que si la consulta principal debe utilizar un cálculo complejo en más de una ocasión, ganamos en eficiencia en la ejecución de las consultas.

3) Construir consultas recursivas: esta característica es de las más importantes, ya que nos permite utilizar SQL para, entre otros, construir jerarquías de datos y, en general, resolver problemas complejos que requieran de recursividad.

Consulta recursiva

Una consulta recursiva es aquella consulta que permite referenciarse a sí misma, implementándose mediante la técnica de CTE. Se verán con posteridad en la siguiente sección.

2.3. Construcción de CTE en PostgreSQL

La posibilidad de crear consultas CTE en PostgreSQL se ha introducido en la versión PostgreSQL 8.4, y la construcción de estas se realiza de la siguiente manera:

```
WITH [RECURSIVE] alias_1 [ ( column_1, column_2, ... ] ) AS (
    query_1
),   [RECURSIVE] alias_2 [ ( column_1, column_2, ... ] ) AS (
    query_2
),
...
     [RECURSIVE] alias_n [ ( column_1, column_2, ... ] ) AS (
    query_n
)
main_query
```

Inicialmente se declaran las **consultas auxiliares** que queremos definir utilizando la cláusula `WITH`, cada una de ellas con un alias específico. Podremos definir tantas consultas auxiliares como sean necesarias, y estas pueden referenciarse unas a otras, siempre y cuando la consulta referenciada esté declarada antes que la consulta que referencia. Es decir, la consulta definida como `alias_n` solamente puede hacer referencia a las consultas definidas anteriormente (desde `alias_1` a `alias_n-1`). Por último, se define lo que denominamos **consulta principal** (`main_query`).

DML

Recordad que DML significa *data manipulation language*.

En PostgreSQL, es importante destacar que las consultas definidas en el `WITH` solamente se ejecutarán si son referenciadas en la consulta principal. Además, tanto las consultas auxiliares definidas como parte de la cláusula `WITH` como la consulta principal (`main_query`) pueden ser tanto expresiones `SELECT` como expresiones DML (`INSERT`, `UPDATE` o `DELETE`).

2.3.1. Consultas recursivas

La cláusula opcional `RECURSIVE` se utiliza para indicar a PostgreSQL que se trata de una consulta recursiva, es decir, que la consulta definida mediante la cláusula `WITH` puede referenciarse a sí misma.

Las consultas recursivas en PostgreSQL, en su forma más simple, se implementan de la siguiente forma:

```
WITH RECURSIVE alias [ ( column_1, column_2, ... ] ) AS (
    non_recursive_term
    [ UNION | UNION ALL]
    recursive_term
)
SELECT expression FROM alias
```

Las partes que la forman son:

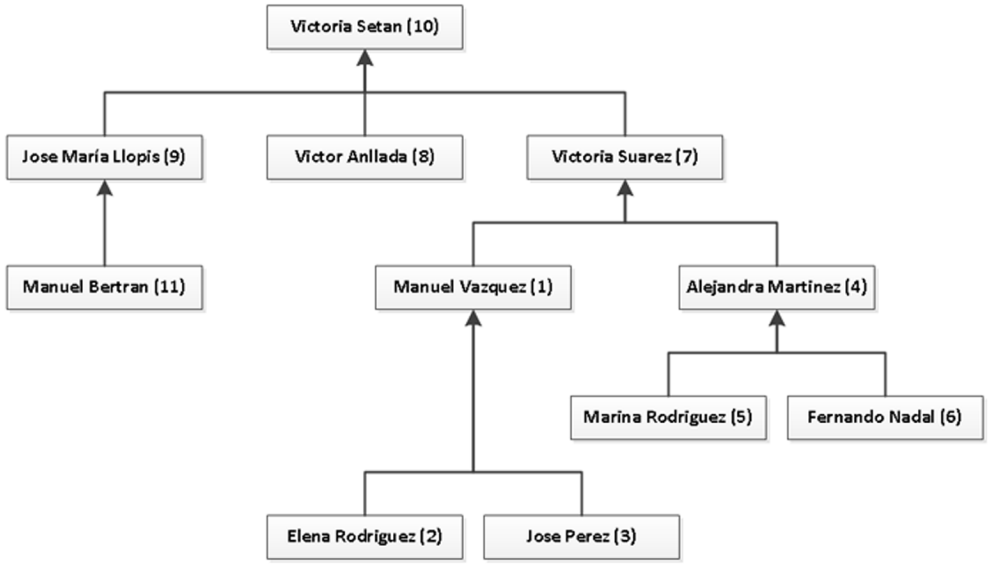
1) Declaración de la consulta recursiva mediante la cláusula `RECURSIVE`, un alias y, opcionalmente, una lista de columnas que corresponderán con la lista de columnas resultado de la consulta.
2) Declaración de la parte no recursiva (`non_recursive_term`), que nunca podrá referenciar a la consulta recursiva.
3) `UNION` o `UNION ALL`, dependiendo de las necesidades: con `UNION` eliminamos los duplicados, con `UNION ALL` los mantenemos.
4) Declaración de la parte recursiva (`recursive_term`), que sí podrá referenciar a la consulta recursiva, y es la que nos permite realizar la recursión (iteración).

Para entender la declaración y funcionamiento de estas consultas, vamos a ver el siguiente ejemplo.

Ejemplo de consulta recursiva: jerarquía de empleados

Supongamos que queremos obtener, para cada uno de los empleados, la estructura jerárquica en la empresa desde el director general hasta el susodicho empleado; esto es, el nombre del director general, el nombre del subordinado, el nombre del siguiente subordinado, etc. y así hasta llegar al empleado en cuestión. Para facilitar el ejemplo, se muestra en la figura siguiente la jerarquía de los datos de empleado de los que disponemos en el ejemplo.

Figura 1. Jerarquía de empleados



Queremos que el resultado de la consulta nos devuelva, para cada empleado, una columna con el siguiente formato:

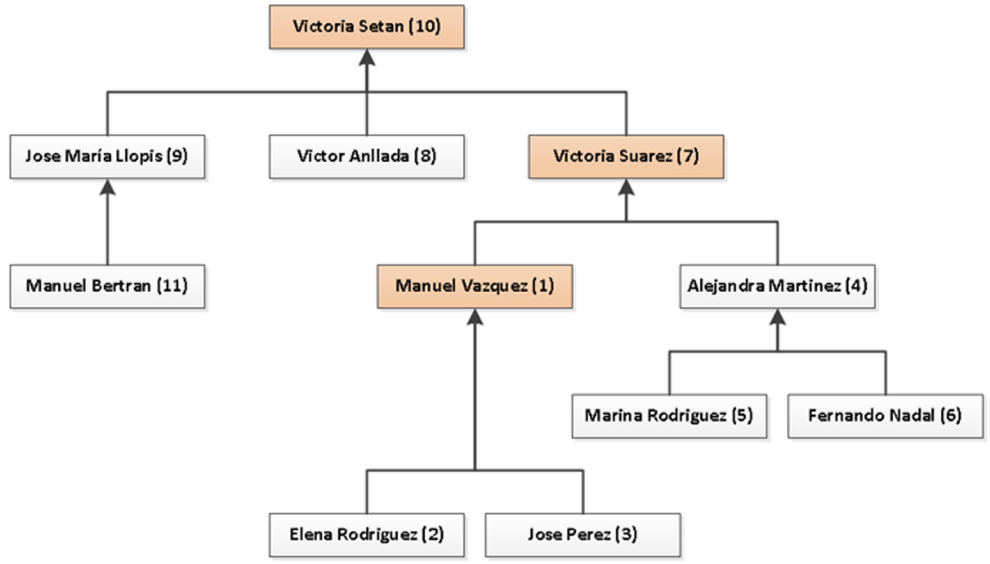
```
Director General <- Empleado Nivel 1 <- Empleado Nivel 2 <- ...
```

Concretamente, y utilizando como ejemplo al empleado Manuel Vázquez (con identificador 1), la jerarquía asociada sería la siguiente:

```
Victoria Setan <- Victoria Suarez <- Manuel Vázquez
```

En la figura 2, podemos ver dónde se encuentra dicho empleado y cuál es la jerarquía asociada a este (celdas resaltadas):

Figura 2. Jerarquía de supervisores para Manuel Vázquez



Realizar esta consulta mediante SQL y sin utilizar CTE supondría realizar dos operaciones de combinación (`JOIN`) sobre la tabla de empleados: la primera para obtener al superior de Manuel Vázquez (que es Victoria Suárez), y la segunda para obtener el superior de Victoria Suárez (que es Victoria Setan, director general). La consulta se implementaría de la siguiente manera:

```
SELECT
    e3.nombre || ' <- ' || e2.nombre || ' <- ' || e1.nombre
FROM
    empleado e1
      INNER JOIN empleado e2
        ON e1.id_supervisor = e2.id_empleado
      INNER JOIN empleado e3
        ON e2.id_supervisor = e3.id_empleado
WHERE
    e1.nombre = 'Manuel Vázquez'
```

Como podéis ver, esto es muy poco eficiente. En el caso de que la jerarquía tuviese más de tres niveles, la consulta tendría que ser modificada para añadir más operaciones de combinación. Además, en este caso concreto sabemos de antemano los niveles que existen entre Manuel Vázquez y el director general, que son tres. ¿Qué pasaría si no dispusiésemos de esta información? Tendríamos que ir probando consultas hasta llegar al punto en el que, subiendo en la jerarquía, obtenemos un empleado con un identificador de supervisor `NULL`, que identifica al director general. Aún más: ¿qué tendríamos que hacer si nos pidiesen estos mismos datos para todos los empleados de la empresa? La solución utilizando el mecanismo anterior sería inviable.

La solución a este tipo de consultas es el uso de consultas recursivas. Suponiendo que tenemos que presentar la jerarquía para todos los empleados de la empresa, la consulta que nos proporciona los resultados deseados sería la que se muestra a continuación:

```
WITH RECURSIVE jerarquias AS (
    SELECT
      id_empleado,
      nombre,
      id_supervisor,
      CAST (nombre AS TEXT) AS resultado
    FROM
      empleado
    WHERE
      id_supervisor IS NULL
    UNION ALL
    SELECT
      e.id_empleado,
      e.nombre,
      e.id_supervisor,
      CAST (j.resultado || ' <- ' || e.nombre AS TEXT) AS resultado
    FROM
      empleado e INNER JOIN jerarquias j
           ON ( e.id_supervisor = j.id_empleado )
 )
SELECT
    nombre,
    resultado
FROM
    jerarquias
ORDER BY
    resultado
```

Destacamos los siguientes aspectos:

- Se puede ver que la parte no recursiva declara una condición `id_supervisor IS NULL`. Esta es la condición que nos permite identificar al último empleado de la jerarquía (el director general).

- Se ha utilizado en este caso `UNION ALL`. La razón principal es para evitar la eliminación de duplicados (aunque es bastante improbable que dos personas con el mismo nombre compartan la misma jerarquía).

- Es importante destacar que la función `CAST` se utiliza para poder almacenar, en una columna denominada resultado, todos los datos de la jerarquía. El `CAST` se realiza a un tipo de dato `TEXT`, ya que desconocemos la longitud máxima en número de caracteres que el formato deseado nos va a ocupar. En el caso de que no se realice, y el valor de dicha columna exceda el de la longitud máxima de la columna nombre (que es la que se utiliza para obtener la jerarquía, siendo de cien caracteres), nos dará un error:

```
ERROR:  recursive query "jerarquias" column 4 has type character varying(100) in non-
recursive term but type character varying overall

LINE 3: (SELECT id_empleado, nombre, id_supervisor, nombre As resultado
                                                    ^
HINT:  Cast the output of the non-recursive term to the correct type.

********** Error **********

ERROR: recursive query "jerarquias" column 4 has type character varying(100) in non-
recursive term but type character varying overall
SQL state: 42804
Hint: Cast the output of the non-recursive term to the correct type.
Character: 72
```

Evaluación de consultas recursivas

¿Cómo evalúa PostgreSQL las consultas recursivas? Estos son los pasos que este SGBD sigue:

- Primero, se evalúa la parte no recursiva (`non_recursive_term`). Estos datos se almacenan en dos lugares: por un lado, en el espacio destinado a los resultados de la consulta `WITH`, que denominaremos **tabla intermedia**, y por otro lado, en una tabla denominada **tabla de trabajo** (*working table* en inglés).

- A continuación, se realizan los siguientes pasos de forma iterativa:
  1. Se evalúa la parte recursiva (`recursive_term`), sustituyendo la llamada a la consulta recursiva por la tabla de trabajo.
  2. A los resultados de evaluar la parte recursiva se les aplica `UNION` o `UNION ALL` con los datos existentes en la tabla intermedia, según se haya especificado en la consulta. En el caso de utilizar `UNION`, se eliminarán los duplicados.
  3. Por último, estos resultados obtenidos se almacenan en la tabla de trabajo, eliminando previamente los resultados de la evaluación anterior.

El proceso iterativo finaliza cuando la ejecución del paso 1 no devuelve ningún resultado.

Veamos un sencillo ejemplo para entender su funcionamiento.

Ejemplo de funcionamiento de consulta recursiva

Para este ejemplo utilizaremos los datos de la tabla de empleados y la consulta recursiva presentados en el ejemplo anterior. Para entender la evaluación de las consultas recursivas, seguimos los pasos definidos anteriormente:

- Evaluamos la parte no recursiva y almacenamos los datos en la tabla intermedia y la tabla de trabajo. Los resultados que contienen ambas tablas se pueden ver a continuación:

Tabla Intermedia			
Id Empleado	**Nombre**	**Id Supervisor**	**Resultado**
10	Victoria Setan		Victoria Setan

Tabla de Trabajo			
Id Empleado	**Nombre**	**Id Supervisor**	**Resultado**
10	Victoria Setan		Victoria Setan

- A continuación, comenzamos con el proceso iterativo.

Iteración 1

Se evalúa la parte recursiva sustituyendo la llamada a la consulta recursiva por los datos que se almacenan en la tabla de trabajo. Los resultados de esta sustitución y evaluación son los siguientes.

Parte Recursiva			
Id Empleado	**Nombre**	**Id Supervisor**	**Resultado**
7	Victoria Suarez	10	Victoria Setan <- Victoria Suarez
8	Víctor Anllada	10	Victoria Setan <- Víctor Anllada
9	José María Llopis	10	Victoria Setan <- José María Llopis

Ved que lo que se está obteniendo en esta parte de la iteración son los empleados que tienen como supervisor a aquellos empleados que existen en la tabla de trabajo.

A continuación, se aplica la cláusula `UNION ALL` (tal y como se ha especificado en la consulta) entre estos resultados y los existentes en la tabla intermedia, guardándolos en esta última. Como último paso, se eliminan los datos de la tabla de trabajo y se añaden los obtenidos en este paso. Al finalizar esta iteración, la tabla intermedia y la de trabajo aparecen ahora con los siguientes datos:

Tabla Intermedia			
Id Empleado	**Nombre**	**Id Supervisor**	**Resultado**
10	Victoria Setan		Victoria Setan
7	Victoria Suarez	10	Victoria Setan <- Victoria Suarez
8	Víctor Anllada	10	Victoria Setan <- Víctor Anllada
9	José María Llopis	10	Victoria Setan <- José María Llopis

Tabla de Trabajo			
Id Empleado	**Nombre**	**Id Supervisor**	**Resultado**
7	Victoria Suarez	10	Victoria Setan <- Victoria Suarez
8	Víctor Anllada	10	Victoria Setan <- Víctor Anllada
9	José María Llopis	10	Victoria Setan <- José María Llopis

Iteración 2

De nuevo se evalúa la parte recursiva, al igual que la iteración anterior, proporcionando los siguientes resultados.

Parte Recursiva			
Id Empleado	**Nombre**	**Id Supervisor**	**Resultado**
1	Manuel Vázquez	7	Victoria Setan <- Victoria Suarez <- Manuel Vázquez
4	Alejandra Martínez	7	Victoria Setan <- Victoria Suarez <- Alejandra Martínez
11	Manuel Bertrán	9	Victoria Setan <- José María Llopis <- Manuel Bertrán

Se obtienen, por lo tanto, los empleados que tienen como supervisor a Victoria Suárez, Víctor Anllada o José María Llopis, que son los empleados que aparecen ahora en la tabla de trabajo.

A continuación, se aplica de nuevo `UNION ALL` entre los resultados obtenidos en esta iteración y los existentes en la tabla intermedia, guardándose estos en esta última. Como último paso, se eliminan los datos de la tabla de trabajo y se añaden estos resultados, dejando la tabla intermedia y de trabajo en el siguiente estado:

Tabla Intermedia			
Id Empleado	**Nombre**	**Id Supervisor**	**Resultado**
10	Victoria Setan		Victoria Setan
7	Victoria Suarez	10	Victoria Setan <- Victoria Suarez
8	Víctor Anllada	10	Victoria Setan <- Víctor Anllada
9	José María Llopis	10	Victoria Setan <- José María Llopis
1	Manuel Vázquez	7	Victoria Setan <- Victoria Suarez <- Manuel Vázquez
4	Alejandra Martínez	7	Victoria Setan <- Victoria Suarez <- Alejandra Martínez
11	Manuel Bertrán	9	Victoria Setan <- José María Llopis <- Manuel Bertrán

Tabla de Trabajo			
Id Empleado	**Nombre**	**Id Supervisor**	**Resultado**
1	Manuel Vázquez	7	Victoria Setan <- Victoria Suarez <- Manuel Vázquez
4	Alejandra Martínez	7	Victoria Setan <- Victoria Suarez <- Alejandra Martínez
11	Manuel Bertrán	9	Victoria Setan <- José María Llopis <- Manuel Bertrán

Iteración 3

En la siguiente iteración, se aplica exactamente el mismo método que en las anteriores, intentando ahora obtener los empleados que tienen como supervisor a Manuel Vázquez, Alejandra Martínez o Manuel Bertrán, siendo estos los siguientes:

Parte Recursiva			
Id Empleado	**Nombre**	**Id Supervisor**	**Resultado**
2	Elena Rodríguez	1	Victoria Setan <- Victoria Suarez <- Manuel Vázquez <- Elena Rodríguez
3	José Pérez	1	Victoria Setan <- Victoria Suarez <- Manuel Vázquez <- José Pérez
5	Marina Rodríguez	4	Victoria Setan <- Victoria Suarez <- Alejandra Martínez <- Marina Rodríguez
6	Fernando Nadal	4	Victoria Setan <- Victoria Suarez <- Alejandra Martínez <- Fernando Nadal

Volvemos a aplicar `UNION ALL` y guardamos los resultados obtenidos en la tabla intermedia de nuevo. Para finalizar, como ya hemos hecho anteriormente, se eliminan los datos de la tabla de trabajo, añadiendo a esta tabla los resultados obtenidos en esta iteración.

Las tablas intermedia y de trabajo aparecen ahora con los siguientes datos:

Tabla Intermedia			
Id Empleado	**Nombre**	**Id Supervisor**	**Resultado**
10	Victoria Setan		Victoria Setan
7	Victoria Suarez	10	Victoria Setan <- Victoria Suarez
8	Víctor Anllada	10	Victoria Setan <- Víctor Anllada
9	José María Llopis	10	Victoria Setan <- José María Llopis
1	Manuel Vázquez	7	Victoria Setan <- Victoria Suarez <- Manuel Vázquez
4	Alejandra Martínez	7	Victoria Setan <- Victoria Suarez <- Alejandra Martínez
11	Manuel Bertrán	9	Victoria Setan <- José María Llopis <- Manuel Bertrán
2	Elena Rodríguez	1	Victoria Setan <- Victoria Suarez <- Manuel Vázquez <- Elena Rodríguez
3	José Pérez	1	Victoria Setan <- Victoria Suarez <- Manuel Vázquez <- José Pérez
5	Marina Rodríguez	4	Victoria Setan <- Victoria Suarez <- Alejandra Martínez <- Marina Rodríguez
6	Fernando Nadal	4	Victoria Setan <- Victoria Suarez <- Alejandra Martínez <- Fernando Nadal

Tabla de Trabajo			
Id Empleado	**Nombre**	**Id Supervisor**	**Resultado**
2	Elena Rodríguez	1	Victoria Setan <- Victoria Suarez <- Manuel Vázquez <- Elena Rodríguez
3	José Pérez	1	Victoria Setan <- Victoria Suarez <- Manuel Vázquez <- José Pérez
5	Marina Rodríguez	4	Victoria Setan <- Victoria Suarez <- Alejandra Martínez <- Marina Rodríguez
6	Fernando Nadal	4	Victoria Setan <- Victoria Suarez <- Alejandra Martínez <- Fernando Nadal

Iteración 4

Se evalúa de nuevo la parte recursiva, y en este caso, no obtenemos ningún resultado, ya que los empleados que existen en la tabla de trabajo no tienen ningún empleado a su cargo.

Como no hay ninguna fila que procesar, una vez llegados a este punto, la tabla de trabajo se destruye y los resultados que contiene la tabla intermedia serán los resultados a procesar por el `SELECT` de la consulta principal (que en este caso es `SELECT nombre, resultado FROM jerarquías ORDER BY resultado`). La consulta, por lo tanto, devolverá el siguiente conjunto de datos (ved que los resultados han sido ordenados de forma diferente a la forma en que se han obtenido):

Nombre	**Resultado**
Victoria Setan	Victoria Setan
José María Llopis	Victoria Setan <- José María Llopis
Manuel Bertrán	Victoria Setan <- José María Llopis <- Manuel Bertrán
Víctor Anllada	Victoria Setan <- Víctor Anllada
Victoria Suarez	Victoria Setan <- Victoria Suarez
Alejandra Martínez	Victoria Setan <- Victoria Suarez <- Alejandra Martínez
Fernando Nadal	Victoria Setan <- Victoria Suarez <- Alejandra Martínez <- Fernando Nadal
Marina Rodríguez	Victoria Setan <- Victoria Suarez <- Alejandra Martínez <- Marina Rodríguez
Manuel Vázquez	Victoria Setan <- Victoria Suarez <- Manuel Vázquez
Elena Rodríguez	Victoria Setan <- Victoria Suarez <- Manuel Vázquez <- Elena Rodríguez
José Pérez	Victoria Setan <- Victoria Suarez <- Manuel Vázquez <- José Pérez

Cuando se trabaja con consultas recursivas, es muy importante asegurarse de que, en algún momento, la parte recursiva no devuelva ninguna fila o corremos el riesgo de que se entre en un bucle infinito.

Una forma que nos puede ayudar a probar consultas que puedan acabar en un bucle infinito es limitar los resultados de la consulta utilizando la cláusula `LIMIT N` (donde `N` es el número de filas que la consulta puede devolver), siempre y cuando los datos de la consulta principal no se ordenen o se combinen con datos de otras tablas.

Ejemplo de bucle infinito en una consulta recursiva

Supongamos que, en lugar de especificar con un valor nulo la falta de un supervisor, esto se especifica mediante la asignación del identificador del propio empleado. De esta forma, el director general (Victoria Setan) tendría como identificador de supervisor su propio identificador de empleado. Por lo tanto, necesitamos cambiar nuestra consulta por la que se muestra a continuación:

```
WITH RECURSIVE jerarquias AS (
    SELECT
      id_empleado,
      nombre,
      id_supervisor,
      CAST (nombre AS TEXT) AS resultado
    FROM
      empleado
    WHERE
      id_supervisor = 10
    UNION ALL
    SELECT
      e.id_empleado,
      e.nombre,
      e.id_supervisor,
      CAST (j.resultado || ' <- ' || e.nombre AS TEXT) AS resultado
    FROM
      empleado e INNER JOIN jerarquias j
           ON ( e.id_supervisor = j.id_empleado )
 )
SELECT
    nombre,
    resultado
FROM
    jerarquias
ORDER BY
    resultado
```

Si ejecutamos esta consulta, veremos que entraremos en un bucle infinito, provocando que el SGBD nunca nos devuelva un resultado. Si queremos ver si estamos en lo cierto, basta ejecutar la consulta sin la cláusula `ORDER BY resultado`, y añadiendo, por ejemplo, la cláusula `LIMIT 1000`, pudiendo ver lo que la consulta está generando en la tabla intermedia.

Ejercicio

Intentad entender por qué se produce un bucle infinito utilizando los pasos que se han explicado anteriormente. Intentad también proporcionar una solución al problema.

2.3.2. Consultas CTE con sentencias de manipulación de datos

Como ya hemos mencionado anteriormente, PostgreSQL permite definir sentencias de manipulación de datos en consultas CTE, tanto en la definición de consultas auxiliares con `WITH` como en la consulta principal. Esto nos puede ser útil para ejecutar varias operaciones en una misma consulta.

Las sentencias DML que forman parte del `WITH` se ejecutan exactamente una única vez, y siempre hasta ser completadas, independientemente de que la consulta principal haga referencia a los datos devueltos. Esta funcionalidad es diferente a la especificada anteriormente cuando la consulta dentro del `WITH` es un `SELECT`, ya que este se ejecuta solamente si es llamado desde la consulta principal.

Veamos un ejemplo para ver esta funcionalidad.

Ejemplo de funcionamiento consulta CTE con `INSERT` y `DELETE`

Imaginemos que queremos eliminar los empleados con identificador 2 y 3 (Elena Rodríguez y José Pérez respectivamente) porque ya no trabajan en nuestra empresa, y moverlos a una tabla de histórico de empleados (`empleado_hist`). Podemos pensar en realizar estas dos operaciones mediante dos sentencias SQL: la primera, una inserción de dichos empleados en la tabla de histórico, y la segunda, un borrado de estos empleados en la tabla maestra, ambas operaciones como parte de una transacción.

Transacciones

Hablaremos de las transacciones en la última sección de este módulo.

```
START TRANSACTION;

INSERT INTO empleado_hist
SELECT * FROM empleado WHERE id_empleado IN (2, 3);

DELETE FROM empleado WHERE id_empleado IN (2, 3);

COMMIT;
```

Esta misma operación podríamos haberla hecho mediante una consulta CTE:

***Common table expression* en PostgreSQL**

Para más información acerca de las CTE en PostgreSQL 9.3, visitad el enlace siguiente: http://www.postgresql.org/docs/9.3/static/queries-with.html

```
WITH empleados_baja AS (
    DELETE
FROM
    empleado
WHERE
    id_empleado IN (2, 3)
    RETURNING *
)
INSERT INTO empleado_hist
  SELECT
      *
  FROM
      empleados_baja
```

La definición de `DELETE` se encarga de eliminar los empleados que han causado baja, devolviendo estas filas eliminadas mediante la cláusula `RETURNING *`, que son las filas que se devuelven al evaluar `empleados_baja`. Estas filas devueltas son las que se utilizan para la inserción en `empleado_hist`.

Si, en cambio, la sentencia DML dentro de `empleados_baja` no devolviese resultados (mediante `RETURNING`), entonces esta no podría ser referenciada por la consulta principal. Así, la siguiente consulta (ved que hemos eliminado `RETURNING *`) nos devolvería el siguiente error:

```
WITH empleados_baja AS (
    DELETE
      FROM empleado
WHERE
    id_empleado IN (2, 3)
)
INSERT INTO empleado_hist
  SELECT
      *
  FROM empleados_baja
```

```
ERROR:  WITH query "empleados_baja" does not have a RETURNING
clause
LINE 10:       empleados_baja
               ^

********** Error **********

ERROR: WITH query "empleados_baja" does not have a RETURNING clause
SQL state: 0A000
Character: 145
```

3. Funciones analíticas

Los SGBD relacionales han incluido, desde sus inicios, funcionalidad para permitir la agregación de datos. Esta funcionalidad, la cual se implementa mediante cláusulas `GROUP BY` y funciones de agregación como `MAX`, `SUM` y `AVG` entre otras, nos permite presentar una visión de los datos de forma agregada. También nos permiten especificar condiciones de búsqueda una vez agregados los datos mediante la cláusula `HAVING`. Por ejemplo, nos permitiría obtener la media del salario de entre los empleados de la empresa agrupada por el puesto de trabajo de los empleados para obtener el salario medio por puesto. A pesar de esta útil funcionalidad, los SGBD todavía tienen la limitación de visualizar los datos de dos formas: datos en bruto (sin procesar) o datos agregados (mediante las funciones de agregación, como hemos comentado). ¿Cómo podríamos presentar ambos tipos de información de forma conjunta? Para ello, como parte del estándar SQL:1999, se han introducido las **funciones analíticas**.

Las funciones analíticas extienden el lenguaje SQL de manera que nos permiten realizar análisis más complejos a la vez que consultamos los datos en bruto sin necesidad de agregarlos mediante el acceso a los datos de otras filas que forman parte de la consulta. Teóricamente, no hay nada que las funciones analíticas realicen que no pueda realizarse mediante consultas SQL complejas, subconsultas u operaciones de combinación, pero sí nos permiten realizar los mismos cálculos de una manera mucho más sencilla y elegante (menos líneas de código), además de utilizar funcionalidad nativa del SGBD que está implementada para dar un rendimiento más óptimo.

Veremos a continuación con más detalle qué son las funciones analíticas, los beneficios que estas nos aportan y qué tipo de problemas nos solucionan, así como aquellas funciones analíticas más comunes en PostgreSQL.

3.1. Concepto

Las **funciones analíticas** (también denominadas *window functions*) se utilizan para realizar cálculos dentro de un contexto de forma que una fila vea y utilice datos más allá de aquellos pertenecientes a dicha fila.

Definimos **ventana** como el contexto en el que la función analítica debe realizar el cálculo especificado. En otras palabras, define qué otras filas se deben tener en consideración (además de la fila actual).

Las funciones analíticas realizan cálculos sobre un conjunto de filas que, de alguna manera, se relacionan con lo que se denomina la fila actual. Es decir, la función analítica es capaz de acceder a información de otras filas desde la fila que se lee o procesa. Para clarificar esta definición, veamos el siguiente ejemplo:

Ejemplo de función analítica

La siguiente tabla Alumno muestra los datos de un conjunto de alumnos dados de alta en la asignatura *Introducción a las bases de datos* impartida en la UOC.

Alumno					
Id Alumno	**Nombre / Apellidos**	**Ciudad**	**Edad**	**Núm. Libros**	**Núm. Asignaturas**
1	Manuel Vázquez	Barcelona	24	10	7
2	Elena Rodríguez	Barcelona	22	6	3
3	José Pérez	Barcelona	25	4	9
4	Alejandra Martínez	Barcelona	18	11	4
5	Marina Rodríguez	Tarragona	19	9	4
6	Fernando Nadal	Tarragona	21	8	5
7	Victoria Suarez	Tarragona	20	6	8
8	Víctor Anllada	Lleida	23	7	7
9	Felisa Sánchez	Lleida	25	5	2
10	José María Llopis	Lleida	18	5	4
11	Victoria Setan	Lleida	23	5	2
12	Wenceslao Fernández	Lleida	18	6	1

Utilizando estos datos como base, supongamos que queremos obtener un listado de ciudades, alumnos y número de asignaturas ordenado por ciudad y nombre de alumno ascendentemente. Para cada uno de los alumnos, queremos mostrar la media de asignaturas que tienen matriculados todos aquellos alumnos que residen en la misma ciudad que el alumno en cuestión. Una propuesta de consulta podría ser la siguiente:

```
SELECT
  a1.ciudad,
  a1.nombre_apellidos,
  a1.num_asignaturas,
  a3.media
FROM
  alumno a1,
  (
    SELECT
      a2.ciudad,
      AVG(a2.num_asignaturas) AS media
    FROM
      alumno a2
    GROUP BY
      a2.ciudad
  ) a3
WHERE
  a1.ciudad = a3.ciudad
ORDER BY
  a1.ciudad ASC,
  a1.nombre_apellidos ASC
```

Los resultados de dicha consulta se pueden ver en la siguiente tabla. Como podéis ver en la consulta propuesta, tenemos que crear una subconsulta sobre la tabla Alumno (denominada a3) que calcule primero la media de asignaturas por ciudad, y realizar a continuación una operación de combinación (JOIN) entre a3 y la tabla Alumno (denominada a1).

Ciudad	Nombre / Apellidos	Núm. Asignaturas	Media
Barcelona	Alejandra Martínez	4	5.75
Barcelona	Elena Rodríguez	3	5.75
Barcelona	José Pérez	9	5.75
Barcelona	Manuel Vázquez	7	5.75
Lleida	Felisa Sánchez	2	3.2
Lleida	José María Llopis	4	3.2
Lleida	Víctor Anllada	7	3.2
Lleida	Victoria Setan	2	3.2
Lleida	Wenceslao Fernández	1	3.2
Tarragona	Fernando Nadal	5	5.666666667
Tarragona	Marina Rodríguez	4	5.666666667
Tarragona	Victoria Suarez	8	5.666666667

Esta forma de crear consultas, si bien nos devuelve los resultados esperados, requiere más procesamiento de datos por parte del SGBD, ya que tiene que calcular primero la media por ciudad para luego combinarla con la misma tabla Alumno. Además, codificar, interpretar y mantener consultas de este estilo puede llegar a ser bastante engorroso.

Veamos lo que las funciones analíticas nos permiten hacer. La siguiente consulta realiza la misma operación que la consulta anterior sin necesidad de crear subconsultas.

```
SELECT
  ciudad,
  nombre_apellidos,
  num_asignaturas,
  AVG(num_asignaturas) OVER (PARTITION BY ciudad) AS media
FROM
  alumno
ORDER BY
  ciudad ASC,
  nombre_apellidos ASC
```

Los resultados, como se esperaba, son los mismos que la consulta anterior, con la diferencia de que el código es mucho más legible y no requiere tanto esfuerzo a la hora de codificar la consulta, además de ser más eficiente a la hora de procesar los datos.

Tras el ejemplo visto, podemos profundizar un poco más en este tipo de funciones. Como se ha visto en el ejemplo, la llamada a funciones analíticas se realiza utilizando un formato especial, que de forma genérica, se podría definir de la forma siguiente:

```
function_name (arg) OVER (
                          [PARTITION BY ...]
                          [ORDER BY ...    ]
                          [ [ROWS | RANGE] ... ]
)
```

La cláusula `OVER` es lo que nos permite distinguir si se están aplicando funciones analíticas o funciones de agregación. Esta cláusula nos permite definir cómo dividir y ordenar el conjunto de datos para su posterior procesamiento por la función.

Esta división de los datos se realiza mediante la cláusula `PARTITION BY`, que crea lo que anteriormente definimos como **ventana**, es decir, una serie de particiones que serán tratadas de forma separada por la función analítica. Para cada fila dentro de cada partición se aplica la función analítica, teniendo en consideración al resto de filas que pertenecen a dicha partición para la realización del cálculo deseado.

También se puede controlar el orden de procesamiento de las filas en cada partición mediante la cláusula `ORDER BY`. Esta ordenación de los datos **dentro de la partición** no afecta (y no tiene que ser la misma que) la **ordenación en la que los datos finales se muestran** tras la ejecución de la consulta.

Las cláusulas mencionadas anteriormente, según sea necesario, se pueden omitir de la llamada:

- En el caso de que la cláusula `PARTITION BY` no se defina, se considerarán todas las filas del resultado de la consulta como una sola partición.
- Cuando se omita la cláusula `ORDER BY`, las filas se procesarán sin un orden definido, lo que podría afectar al resultado final, dependiendo de la función analítica que se utilice.

Otro concepto relevante a considerar, cuando trabajamos con funciones analíticas, es el de **marco**, que definimos a continuación.

El **marco** (*frame*) se define como un subconjunto de filas dentro de una partición. Muchas de las funciones analíticas (no todas) actúan dentro del marco definido en lugar de actuar sobre el conjunto de datos de la partición.

En otras palabras, el marco permite definir una **subventana dentro de la ventana** establecida mediante la cláusula `PARTITION BY`. El marco permite cambiar el contexto de la ventana de forma **dinámica** (el marco incrementa/reduce las filas a tener en cuenta en relación con la fila que se procese) o bien establecer un contexto de ventana **estático** (un límite inferior y un límite superior fijos). El marco se puede definir mediante las cláusulas `RANGE` o `ROWS`.

Suma acumulativa

Definimos como **suma acumulativa** (*running sum*, *cumulative sum* o *running total* en inglés) como la suma en secuencia de números, cuyo valor se actualiza cada vez que un número se añade a la secuencia, sumando dicho número a la suma acumulativa previa.

Diferencia entre ventana y marco

Supongamos que tenemos la siguiente consulta que nos proporciona la **suma acumulativa** de las asignaturas de los alumnos que viven en cada ciudad. La consulta SQL necesaria sería la siguiente.

```
SELECT
  ciudad,
  nombre_apellidos,
  num_asignaturas,
  SUM(num_asignaturas) OVER
    (PARTITION BY ciudad ORDER BY nombre_apellidos ASC
     RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS suma
FROM
  alumno
ORDER BY
  ciudad ASC,
  nombre_apellidos ASC
```

Los resultados obtenidos se pueden ver en la tabla inferior.

Ciudad	Nombre / Apellidos	Núm. Asignaturas	Suma
Barcelona	Alejandra Martínez	4	4
Barcelona	Elena Rodríguez	3	7
Barcelona	José Pérez	9	16
Barcelona	Manuel Vázquez	7	23
Lleida	Felisa Sánchez	2	2
Lleida	José María Llopis	4	6
Lleida	Víctor Anllada	7	13
Lleida	Victoria Setan	2	15
Lleida	Wenceslao Fernández	1	16
Tarragona	Fernando Nadal	5	5
Tarragona	Marina Rodríguez	4	9
Tarragona	Victoria Suarez	8	17

En este caso, la **ventana** de la consulta se define a partir de *CIUDAD*, mediante la cláusula `PARTITION BY CIUDAD`, obteniendo así tres particiones de datos: los que viven en Barcelona, los que viven en Lleida y aquellos que viven en Tarragona.

Cada una de estas particiones se ordena alfabética y ascendentemente por nombre de alumno (cláusula `ORDER BY NOMBRE_APELLIDOS ASC` como parte de `OVER`). No debemos confundir con cómo los datos de salida se ordenan (`ORDER BY CIUDAD ASC, NOMBRE_APELLIDOS ASC`).

Por último, el marco se ha definido como `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. Esto significa que el marco se define desde el inicio de la partición hasta la fila actual. Utilizando la partición de Barcelona como ejemplo, esto significa que:

- Para el alumno Alejandra Martínez el marco incluye solamente esta fila (es la primera fila de la partición, y como límite superior es la fila actual), por lo que la suma acumulativa incluye solamente el número de asignaturas de este alumno.

Marco del alumno "Alejandra Martínez"			
Ciudad	**Nombre / Apellidos**	**Núm. Asignaturas**	**Suma**
Barcelona	Alejandra Martínez	4	4

- Para el alumno Elena Rodríguez el marco incluye desde el inicio de la partición (Alejandra Martínez) hasta la fila actual que se procesa (Elena Rodríguez), por lo que la suma acumulativa incluye el número de asignaturas de este alumno más las de Alejandra Martínez, sumando un total de 7 asignaturas (4 + 3).

Marco del alumno "Elena Rodríguez"			
Ciudad	**Nombre / Apellidos**	**Núm. Asignaturas**	**Suma**
Barcelona	Alejandra Martínez	4	4
Barcelona	Elena Rodríguez	3	7

- Para el alumno José Pérez el marco incluye desde el inicio de la partición (Alejandra Martínez) hasta la fila actual que se procesa (José Pérez), por lo que la suma acumulativa incluye el número de asignaturas de este alumno más las de Elena Rodríguez y Alejandra Martínez, sumando un total de 16 asignaturas (4 + 3 + 9).

Marco del alumno "José Pérez"			
Ciudad	**Nombre / Apellidos**	**Núm. Asignaturas**	**Suma**
Barcelona	Alejandra Martínez	4	4
Barcelona	Elena Rodríguez	3	7
Barcelona	José Pérez	9	16

- Por último, para el alumno Manuel Vázquez el marco incluye desde el inicio de la partición (Alejandra Martínez) hasta la fila actual que se procesa (Manuel Vázquez), por lo que la suma acumulativa incluye el número de asignaturas de este alumno más las de José Pérez, Elena Rodríguez y Alejandra Martínez, sumando un total de 16 asignaturas (4 + 3 + 9 + 7).

Marco del alumno "Manuel Vázquez"			
Ciudad	**Nombre / Apellidos**	**Núm. Asignaturas**	**Suma**
Barcelona	Alejandra Martínez	4	4
Barcelona	Elena Rodríguez	3	7
Barcelona	José Pérez	9	16
Barcelona	Manuel Vázquez	7	23

Para terminar, es importante destacar que las funciones analíticas solamente pueden llamarse como parte de las cláusulas `SELECT` y `ORDER BY` dentro de una consulta, es decir, no se permite su utilización en cláusulas `WHERE`, `GROUP BY` o `HAVING`, debido a que las funciones analíticas se procesan después de que dichas cláusulas se hayan completado.

3.2. Beneficios de las funciones analíticas

Entre los posibles beneficios del uso de funciones analíticas se destacan los siguientes:

1) Facilitar la obtención de cálculos complejos en informes y procesos ETL de forma más sencilla, complejidad que suele darse muy a menudo dentro del ámbito de los *data warehouse*.

2) Mejorar el rendimiento de las consultas SQL: consultas que antes requerían operaciones de combinación sobre la misma tabla se pueden implementar con cláusulas SQL mucho más sencillas.

3) Proporcionar una manera más clara y concisa de generar consultas SQL, lo que facilita el mantenimiento del código e incrementa la productividad de los desarrolladores.

4) La sintaxis de funciones analíticas forma parte de SQL estándar, lo que significa que están soportadas por multitud de SGBD del mercado, entre ellos PostgreSQL y Oracle.

Sobre la base de lo comentado, y a modo de resumen, podemos afirmar que las funciones analíticas nos facilitan el cálculo, de forma eficiente y elegante, sobre un conjunto de filas para devolvernos un valor relacionado con un subconjunto de datos de dicha consulta.

A continuación, vamos a ver cómo se realizan las llamadas de funciones analíticas en PostgreSQL, así como la sintaxis necesaria y sus diferentes reglas.

3.3. Funciones analíticas en PostgreSQL

La llamada a funciones analíticas en PostgreSQL se puede realizar utilizando cualquiera de las siguientes formas:

```
function_name ([expression [, expression ... ]]) OVER (window_definition)
function_name ( * ) OVER (window_definition)
```

En estas definiciones, `expression` representa cualquier expresión que no contenga una llamada a una función analítica: podría tratarse de una columna de una tabla, una función de agregación, una constante o un cálculo, entre otros.

La cláusula `window_definition` nos permite definir la ventana mediante la siguiente sintaxis:

```
[ existing_window_name ]

[ PARTITION BY expression [, ...] ]

[ ORDER BY expression [ ASC | DESC | USING operator ]
                     [ NULLS { FIRST | LAST } ] [, ...] ]

[ frame_clause ]
```

Como hemos visto anteriormente, las cláusulas `PARTITION BY` y `ORDER BY` nos sirven para definir las particiones y cómo los datos dentro de cada partición serán ordenados. Por su parte, la cláusula opcional `frame_clause` permite trabajar a la función analítica dentro de un **marco**. Esta cláusula se puede especificar mediante cualquiera de las dos siguientes opciones:

```
{ RANGE | ROWS } frame_start

{ RANGE | ROWS } BETWEEN frame_start AND frame_end
```

donde los posibles valores para los parámetros`frame_start` y `frame_end`, que delimitan el ámbito del marco, pueden ser alguna de las siguiente opciones:

```
UNBOUNDED PRECEDING

offset PRECEDING

CURRENT ROW

offset FOLLOWING

UNBOUNDED FOLLOWING
```

Como se ha comentado anteriormente, el marco puede especificarse mediante rango (`RANGE`) o mediante filas (`ROWS`). `RANGE` permite definir rangos de filas utilizando los delimitadores `UNBOUNDED PRECEDING`, `CURRENT ROW` y `UNBOUNDED FOLLOWING`, sin poder acceder a una posición específica dentro de la partición. De esto último se encarga `ROWS`, esto es, permite no solo especificar rangos mediante los delimitadores que permite `RANGE`, sino que además permite definir marcos utilizando posiciones concretas dentro de la partición mediante la definición de un *offset*.

Offset

Distancia o posición relativa a un elemento.

Si el parámetro `frame_start` se define como `UNBOUNDED PRECEDING` entonces significa que el marco comienza con la primera fila de la partición, y de forma similar, si `frame_end` se define como `UNBOUNDED FOLLOWING` enton-

ces significa que el marco acaba en la última fila de la partición. Si `frame_end` no se especifica, entonces el valor por defecto es la fila que se está procesando (`CURRENT ROW`). Por defecto, en el caso de que se omita la definición del marco, la opción seleccionada por PostgreSQL es `RANGE UNBOUNDED PRECEDING`, que es equivalente a especificar `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`.

Si se especifica `ORDER BY` en la definición de la ventana, entonces el marco se establece desde el inicio de la partición hasta la última fila representativa del conjunto de filas equivalente definido por `ORDER BY`. Si no se especifica `ORDER BY`, todas las filas de la partición se incluyen dentro del marco, ya que todas se consideran representativas o pares a la fila procesada.

En modo `RANGE`, si el parámetro `frame_start` se define como `CURRENT ROW` esto significa que el marco comienza con la primera fila representativa del conjunto de filas equivalente que define la cláusula `ORDER BY`, mientras que si el parámetro `frame_end` se define como `CURRENT ROW` esto significa que el marco termina en la última fila representativa del conjunto de filas equivalente que define la cláusula `ORDER BY`.

Ejemplo de `RANGE`

Como ejemplo de uso de `RANGE`, tenemos la siguiente consulta utilizada anteriormente, que nos proporciona la suma acumulativa de las asignaturas de los alumnos que viven en cada ciudad. Esta consulta utiliza la cláusula `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`, lo que nos indica que el marco de la fila se establece desde el inicio de la partición hasta la fila actual.

```
SELECT
  ciudad,
  nombre_apellidos,
  num_asignaturas,
  SUM(num_asignaturas) OVER
    (PARTITION BY ciudad ORDER BY nombre_apellidos ASC
     RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS suma
FROM
  alumno
ORDER BY
  ciudad ASC,
  nombre_apellidos ASC
```

En modo `ROWS`, cuando se especifica `CURRENT ROW` en la definición del marco, significa que el marco comienza o acaba con la fila actual. En cambio, en modo `RANGE` significa que el marco comienza o acaba con la primera o última fila del conjunto de filas representativas según estén ordenadas mediante `ORDER BY`.

Es importante destacar que el uso de la cláusula `ROWS` podría producir resultados inesperados si la cláusula `ORDER BY` no ordena las filas de forma única. Las opciones de `RANGE` están precisamente diseñadas para asegurar que las filas emparejadas sobre la base del `ORDER BY` sean tratadas de forma similar: cualquier par de filas que estén emparejadas pertenecerán o no al marco.

Las cláusulas `offset PRECEDING` y `offset FOLLOWING` solamente están disponibles en modo `ROWS`. Esto indica que el marco comienza con la fila que se encuentra en la posición *offset* antes de la fila actual (`offset PRECEDING`)

y termina con la fila que se encuentra en la posición *offset* después de la fila actual (`offset FOLLOWING`). El valor de *offset* debe ser una expresión entera que no contenga variables, ni funciones de agregación ni analíticas. El valor tampoco puede ser nulo o negativo, si bien puede ser cero, indicando en este caso que se trata de la fila actual (`CURRENT ROW`).

Ejemplo de uso de cláusula `ROWS` con *offset*

Utilizaremos como base la consulta de suma acumulativa de asignaturas que hemos visto hasta ahora. La nueva consulta definida utiliza la cláusula `ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING`, lo que nos indica que el marco de la fila se establece desde una fila anterior a la fila actual hasta una fila posterior a la fila actual (en este caso, ambos *offset* se han especificado como 1).

```
SELECT
  ciudad,
  nombre_apellidos,
  num_asignaturas,
  SUM(num_asignaturas) OVER (PARTITION BY ciudad
        ORDER BY nombre_apellidos ASC
            ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS suma
FROM
  alumno
ORDER BY
  ciudad ASC,
  nombre_apellidos ASC
```

Los resultados de esta consulta serán los siguientes. La suma que aparece en cada fila se obtiene mediante la suma de las asignaturas del alumno inmediatamente anterior a la fila actual, al alumno inmediatamente posterior a la fila actual, y las asignaturas de la fila actual, siempre dentro de la partición establecida (CIUDAD).

Ciudad	Nombre / Apellidos	Núm. Asignaturas	Suma
Barcelona	Alejandra Martínez	4	7
Barcelona	Elena Rodríguez	3	16
Barcelona	José Pérez	9	19
Barcelona	Manuel Vázquez	7	16
Lleida	Felisa Sánchez	2	6
Lleida	José María Llopis	4	13
Lleida	Víctor Anllada	7	13
Lleida	Victoria Setan	2	10
Lleida	Wenceslao Fernández	1	3
Tarragona	Fernando Nadal	5	9
Tarragona	Marina Rodríguez	4	17
Tarragona	Victoria Suarez	8	12

Tomando como ejemplo el alumno Elena Rodríguez: la suma de las asignaturas es 16, porque es la suma de las asignaturas de Alejandra Martínez (fila anterior, 4 asignaturas), José Pérez (fila posterior, 3 asignaturas) y Elena Rodríguez (fila actual, 3 asignaturas).

Existen una serie de restricciones sobre la definición de marcos: `frame_start` no puede tomar un valor `UNBOUNDED FOLLOWING` (es decir, no puede comenzar con la última fila de la partición), `frame_end` no puede tomar un valor

UNBOUNDED PRECEDING (es decir, no puede finalizar con la primera fila de la partición), y la opción seleccionada como frame_end no puede referirse a una fila que aparezca antes que el valor de frame_start (como ejemplo, la opción RANGE BETWEEN CURRENT ROW AND value PRECEDING no estaría permitida).

En el caso de que se use el formato de ventana existing_window_name, este ha de referirse a una entrada en la lista de ventanas especificada mediante la cláusula WINDOW. Las llamadas a funciones analíticas con ventanas definidas mediante WINDOW se realizan de la siguiente manera:

```
function_name ([expression [, expression ... ]]) OVER window_name

function_name ( * ) OVER window_name
```

En este caso, window_name referencia a una ventana especificada mediante la cláusula WINDOW de PostgreSQL dentro de la consulta. Uno de los beneficios de usar esta cláusula es que permite referenciar la misma ventana en varias partes de la consulta, de forma que evitamos la duplicidad de cláusulas OVER y así evitar errores en la definición.

El funcionamiento de este tipo de llamadas es el siguiente: se copia la definición de la ventana definida mediante esta cláusula, por lo que la ventana nueva no puede incluir su propio PARTITION BY, pero sí puede especificar su propio ORDER BY si la ventana que se utiliza como plantilla no lo tiene definido. La nueva ventana definida siempre usa su propio frame_clause, por lo que la ventana existente no puede especificar dicha cláusula.

La cláusula WINDOW permite no solamente crear código más legible, sino que además permite reutilizar las ventanas dentro de una misma consulta.

La forma que propone PostgreSQL para definir ventanas mediante la cláusula WINDOW es la siguiente:

```
WINDOW window_name AS ( window_definition ) [, ...]
```

donde window_definition utiliza el mismo formato de definición de ventanas explicado anteriormente.

Es importante destacar la diferencia en el uso de OVER en las dos formas que hemos visto. En la primera forma, se utiliza OVER (window_definition), mientras que en el segundo se utiliza OVER window_name (ved que el primero utiliza paréntesis y el otro no). El primero requiere de forma obligatoria la definición de una ventana, mientras que en el segundo se requiere que dicha ventana esté definida como parte de la cláusula WINDOW.

Nota

Para obtener más detalles acerca de la cláusula WINDOW de PostgreSQL, visitad el siguiente vínculo:

http://www.postgresql.org/docs/9.3/static/sql-select.html

Ejemplo de utilización de la cláusula `WINDOW`

Al principio de esta sección se ha propuesto un ejemplo para obtener un listado de ciudades, alumnos y número de asignaturas ordenado por ciudad y nombre de alumno ascendentemente. Para cada uno de los alumnos, se quería mostrar la media de asignaturas que tienen matriculados todos aquellos alumnos que residen en la misma ciudad que el alumno en cuestión. Una propuesta de consulta, utilizando funciones analíticas, sería la siguiente:

```
SELECT
  ciudad,
  nombre_apellidos,
  num_asignaturas,
  AVG(num_asignaturas) OVER (PARTITION BY ciudad) AS media
FROM
  alumno
ORDER BY
  ciudad ASC,
  nombre_apellidos ASC
```

Esta misma consulta podría definirse de la siguiente forma utilizando la cláusula `WINDOW`:

```
SELECT
  ciudad,
  nombre_apellidos,
  num_asignaturas,
  AVG(num_asignaturas) OVER w AS media
FROM
  alumno
WINDOW w AS (PARTITION BY ciudad)
ORDER BY
  ciudad ASC,
  nombre_apellidos ASC
```

Por último, y para finalizar este apartado sobre funciones analíticas en PostgreSQL, es importante mencionar que este SGBD permite definir sus propias funciones analíticas utilizando alguna de las API que proporciona. Aunque es un tema que puede ser interesante para el estudiante, se ha considerado que está fuera del ámbito de este módulo didáctico.

3.4. Tipos de funciones analíticas en PostgreSQL

En las siguiente subsecciones veremos algunos ejemplos de las funciones analíticas más importantes que implementa PostgreSQL. Para los ejemplos, vamos a utilizar la tabla Alumno con los siguientes datos:

Alumno					
Id Alumno	**Nombre / Apellidos**	**Ciudad**	**Edad**	**Núm. Libros**	**Núm. Asignaturas**
1	Manuel Vázquez	Barcelona	24	10	7
2	Elena Rodríguez	Barcelona	22	6	3
3	José Pérez	Barcelona	25	4	9
4	Alejandra Martínez	Barcelona	18	11	4
5	Marina Rodríguez	Tarragona	19	9	4
6	Fernando Nadal	Tarragona	21	8	5
7	Victoria Suarez	Tarragona	20	6	8

Alumno					
Id Alumno	**Nombre / Apellidos**	**Ciudad**	**Edad**	**Núm. Libros**	**Núm. Asignaturas**
8	Víctor Anllada	Lleida	23	7	7
9	Felisa Sánchez	Lleida	25	5	2
10	José María Llopis	Lleida	18	5	4
11	Victoria Setan	Lleida	23	5	2
12	Wenceslao Fernández	Lleida	18	6	1

3.4.1. *Row number*

Esta función asigna un número único a cada fila dentro de una partición, comenzando desde el valor 1, y de forma secuencial según la especificación de la cláusula `ORDER BY`. Esta función se define como `row_number()` y no acepta parámetros.

Nota

Consultar el siguiente vínculo para obtener una descripción de todas las funciones analíticas presentes en PostgreSQL 9.3:

http://www.postgresql.org/docs/9.3/static/functions-window.html

Ejemplo de utilización de `row_number()`

Utilizando la tabla de alumnos indicada, obtener un listado con el nombre de la ciudad, el nombre y apellidos del alumno, y la posición del alumno en cada ciudad utilizando el nombre y apellidos (alfabéticamente) como criterio de cálculo de posición.

```
SELECT
  ciudad,
  nombre_apellidos,
  ROW_NUMBER() OVER (PARTITION BY ciudad
                     ORDER BY nombre_apellidos ASC) AS rn
FROM
  alumno
ORDER BY
  ciudad ASC,
  nombre_apellidos ASC
```

Los resultados de esta consulta serán los siguientes. Ved que para cada ciudad (`PARTITION BY`), a cada uno de los alumnos se le asigna un número secuencial –la posición, que se obtiene mediante `row_number()`– en base al nombre y los apellidos ordenados alfabéticamente (`ORDER BY`).

Ciudad	Nombre / Apellidos	RN
Barcelona	Alejandra Martínez	1
Barcelona	Elena Rodríguez	2
Barcelona	José Pérez	3
Barcelona	Manuel Vázquez	4
Lleida	Felisa Sánchez	1
Lleida	José María Llopis	2
Lleida	Víctor Anllada	3
Lleida	Victoria Setan	4
Lleida	Wenceslao Fernández	5
Tarragona	Fernando Nadal	1
Tarragona	Marina Rodríguez	2
Tarragona	Victoria Suarez	3

Este tipo de consultas nos puede servir, por ejemplo, para obtener el listado de aquellos alumnos que se encuentran en una posición concreta. Por ejemplo, veamos la consulta necesaria para obtener aquellos alumnos en la posición número 2 (utilizando el mismo criterio especificado anteriormente).

```
SELECT
  ds.ciudad,
  ds.nombre_apellidos
FROM
  (
    SELECT
      ciudad,
      nombre_apellidos,
      ROW_NUMBER() OVER (PARTITION BY ciudad
                         ORDER BY nombre_apellidos ASC) AS rn
    FROM
      alumno
  )
  ds
WHERE
  ds.rn = 2
ORDER BY
  ds.ciudad ASC,
  ds.nombre_apellidos ASC
```

El resultado de esta consulta es el siguiente:

Ciudad	Nombre / Apellidos
Barcelona	Elena Rodríguez
Lleida	José María Llopis
Tarragona	Marina Rodríguez

Véase que, para poder obtener aquellos alumnos en la posición 2, hemos tenido que utilizar subconsultas. La razón principal es que, como se ha comentado anteriormente, no se permiten funciones analíticas como parte de la cláusula `WHERE`, por lo que el uso de

subconsultas es necesario para aplicar condiciones sobre los resultados que nos proporcionan estas funciones.

3.4.2. *Rank*

Esta función realiza un *ranking* de las filas de una partición **con huecos**. En otras palabras, esta función permite clasificar los elementos de un grupo en posiciones (primero, segundo, tercero, etc.), y si hay elementos con el mismo valor los empareja dentro de la misma posición, pero al inmediato inferior le da la posición en base al número de elementos existentes. Esta función se define como `rank()`, y no acepta parámetros.

Ejemplo de utilización de `rank()`

Para este ejemplo, queremos obtener un listado con el nombre de la ciudad, el nombre y apellidos del alumno, y el *ranking* en número de asignaturas de cada uno de los alumnos en cada ciudad. Este *ranking* se ha de obtener de forma que aquellos alumnos con más asignaturas aparezcan con una posición en el *ranking* superior.

```
SELECT
  ciudad,
  nombre_apellidos,
  num_asignaturas,
  RANK() OVER (PARTITION BY ciudad
                ORDER BY num_asignaturas DESC) AS rk
FROM
  alumno
ORDER BY
  ciudad ASC,
  rk ASC
```

Los resultados que obtenemos se muestran en la siguiente tabla: podemos ver, utilizando Lleida como ejemplo, que el alumno con más asignaturas (7) aparece como primero en el *ranking* solamente para dicha ciudad (existe una partición por CIUDAD). Véase también la posición de los alumnos Felisa Sánchez y Victoria Setan. Ambos alumnos tienen 2 asignaturas y un *ranking* asignado de 3, es decir, existe un empate. El siguiente en número de asignaturas es Wenceslao Fernández, con una asignatura y un puesto 5 en el *ranking*.

Ciudad	Nombre / Apellidos	Núm. Asignaturas	RK
Barcelona	José Pérez	9	1
Barcelona	Manuel Vázquez	7	2
Barcelona	Alejandra Martínez	4	3
Barcelona	Elena Rodríguez	3	4
Lleida	Víctor Anllada	7	1
Lleida	José María Llopis	4	2
Lleida	Felisa Sánchez	2	3
Lleida	Victoria Setan	2	3
Lleida	Wenceslao Fernández	1	5
Tarragona	Victoria Suarez	8	1
Tarragona	Fernando Nadal	5	2
Tarragona	Marina Rodríguez	4	3

Puede sorprender que el *ranking* salte de 3 a 5 (como alguno puede pensar). Esta es la característica de la función `rank()`: en el caso de que haya un empate, el *ranking* de la siguiente fila tendrá asignado el número de posición de la fila dentro de dicha partición, dejando un hueco entre el último *ranking* utilizado en el empate y el número de posición de la fila. En este caso, el alumno Wenceslao Fernández es la fila número 5 dentro de la partición de Lleida, y como es un nuevo valor dentro del *ranking*, se le asocia el *ranking* número 5, dejando un hueco (que sería la posición de *ranking* 4).

3.4.3. *Dense rank*

Esta función realiza un *ranking* de las filas de una partición **sin huecos**. Al igual que la función `rank()`, esta función permite clasificar los elementos de un grupo en posiciones (primero, segundo, tercero, etc.). En el caso de que existan elementos con el mismo valor, los coloca dentro de la misma posición (emparejadas) y al inmediato inferior le da el correlativo siguiente a la clasificación de posición. Esta función se define como `dense_rank()`, y no acepta parámetros.

Ejemplo de utilización de `dense_rank()`

En este caso vamos a utilizar el mismo requisito que en el ejemplo de la función `rank()`, salvo que vamos a utilizar la función `dense_rank()`.

```
SELECT
  ciudad,
  nombre_apellidos,
  num_asignaturas,
  DENSE_RANK() OVER (PARTITION BY ciudad
                     ORDER BY num_asignaturas DESC) AS rk
FROM
  alumno
ORDER BY
  ciudad ASC,
  rk ASC
```

Los resultados que obtenemos son los siguientes: se puede ver que el listado obtenido es prácticamente el mismo que en el ejemplo propuesto anteriormente excepto el *ranking* asociado al alumno Wenceslao Fernández. En este caso, este alumno tiene asociado un *ranking* 4 en lugar de 5, como sucedía con la función `rank()`. La razón es que, a diferencia de `rank()`, `dense_rank()` no deja huecos en las posiciones de *ranking*.

Ciudad	Nombre / Apellidos	Núm. Asignaturas	RK
Barcelona	José Pérez	9	1
Barcelona	Manuel Vázquez	7	2
Barcelona	Alejandra Martínez	4	3
Barcelona	Elena Rodríguez	3	4
Lleida	Víctor Anllada	7	1
Lleida	José María Llopis	4	2
Lleida	Victoria Setan	2	3
Lleida	Felisa Sánchez	2	3
Lleida	Wenceslao Fernández	1	4
Tarragona	Victoria Suarez	8	1
Tarragona	Fernando Nadal	5	2
Tarragona	Marina Rodríguez	4	3

3.4.4. *Lag*

Esta función permite acceder a la información almacenada en alguna de las filas **previas** a la fila actual (`CURRENT ROW`) dentro de la partición. Esta función se define como `lag()`, y acepta los siguientes parámetros:

```
lag ( expression [, offset] [, default] );
```

- `expression`: cualquier valor a evaluar excepto funciones analíticas (por ejemplo, una columna de una tabla, una función escalar, etc.).
- `offset` (opcional): indica la posición de la fila previa a la que se va a acceder desde la fila actual en la partición. Por ejemplo, un valor de 3 indica que se va a acceder a la tercera fila previa a la fila actual. Si se omite, por defecto se asigna un valor 1 (la fila anterior).
- `default` (opcional): el valor por defecto a asignar en el caso de que la fila a acceder esté fuera de los límites permitidos. Si se omite, por defecto se asigna un valor `NULL`.

Ejemplo de utilización de `lag()`

Para el siguiente ejemplo, se pide obtener un listado de alumnos (nombre y apellidos), con su edad, y la edad del alumno que se encuentra dos posiciones por detrás, suponiendo

que los alumnos están ordenados de forma alfabética ascendente. En el caso de que no exista información acerca de alumnos previos, se indicará con un valor por defecto de -1.

```
SELECT
  nombre_apellidos,
  edad,
  LAG(edad, 2, -1)
        OVER (ORDER BY nombre_apellidos ASC) AS edad_anterior
FROM
  alumno
ORDER BY
  nombre_apellidos ASC,
  edad_anterior ASC
```

Los resultados que obtenemos son los siguientes: los dos primeros alumnos (Alejandra Martínez y Elena Rodríguez) no tienen alumnos en dos posiciones previas alfabéticamente, por lo que se asigna el valor por defecto especificado en la función (-1). En cambio, el alumno Felisa Sánchez tiene una `EDAD_ANTERIOR = 18`, que es la edad del alumno dos posiciones atrás en orden alfabético (esto es, Alejandra Martínez).

Nombre / Apellidos	Edad	Edad Anterior
Alejandra Martínez	18	-1
Elena Rodríguez	22	-1
Felisa Sánchez	25	18
Fernando Nadal	21	22
José María Llopis	18	25
José Pérez	25	21
Manuel Vázquez	24	18
Marina Rodríguez	19	25
Víctor Anllada	23	24
Victoria Setan	23	19
Victoria Suarez	20	23
Wenceslao Fernández	18	23

Podemos observar en este ejemplo que, a diferencia de los otros expuestos hasta este momento, no se ha utilizado la cláusula `PARTITION BY`. Esto es así porque sobre la base del requisito especificado, todo el conjunto de datos de la tabla de alumnos es considerado como una única partición.

3.4.5. *Lead*

Al contrario que la función `lag()`, esta función permite acceder a la información almacenada en alguna de las filas **posteriores** a la fila actual dentro de la partición. Esta función se define como `lead()`, y acepta los siguientes parámetros:

```
lead ( expression [, offset] [, default] );
```

- `expression`: cualquier valor a evaluar excepto funciones analíticas (por ejemplo, una columna de una tabla, una función escalar, etc.).
- `offset` (opcional): indica la posición de la fila posterior a la que se va a acceder desde la fila actual en la partición. Por ejemplo, un valor de 3 indica que se va a acceder a la tercera fila posterior a la fila actual. Si se omite, por defecto se asigna un valor 1 (la fila siguiente).
- `default` (opcional): el valor por defecto a asignar en el caso de que la fila a acceder esté fuera de los límites permitidos. Si se omite, por defecto se asigna un valor `NULL`.

Ejemplo de utilización de `lead()`

En este ejemplo, utilizaremos el mismo criterio que en el ejemplo utilizado para la función `lag()`, pero en lugar de ser dos posiciones previas, serán dos posiciones posteriores y con el mismo valor por defecto.

```
SELECT
  nombre_apellidos,
  edad,
  LEAD(edad, 2, -1) OVER
      (ORDER BY nombre_apellidos ASC) AS edad_posterior
FROM
  alumno
ORDER BY
  nombre_apellidos ASC,
  edad_posterior ASC
```

Los resultados que obtenemos son los siguientes: para los dos primeros alumnos (Alejandra Martínez y Elena Rodríguez), al contrario de lo que sucedía con `lag()`, sí que tenemos posiciones posteriores. Esto no es así con los dos últimos alumnos (Victoria Suárez y Wenceslao Fernández), de ahí que tengan un valor por defecto de -1.

Nombre / Apellidos	Edad	Edad Posterior
Alejandra Martínez	18	25
Elena Rodríguez	22	21
Felisa Sánchez	25	18
Fernando Nadal	21	25
José María Llopis	18	24
José Pérez	25	19
Manuel Vázquez	24	23
Marina Rodríguez	19	23
Víctor Anllada	23	20
Victoria Setan	23	18
Victoria Suarez	20	-1
Wenceslao Fernández	18	-1

Ambas funciones `lag()` y `lead()` nos sirven para acceder a datos en diferentes posiciones dentro de la partición especificada sin necesidad de realizar operaciones de combinación (`JOIN`) con la misma tabla.

3.4.6. *First value*

Esta función devuelve el valor de una expresión asociado a la primera fila del marco definido en la consulta. Esta función se define como `first_value()`, y contiene el siguiente parámetro:

```
first_value ( expression );
```

- `expression`: cualquier valor a evaluar excepto funciones analíticas (por ejemplo, una columna de una tabla, una función escalar, etc.).

Ejemplo de utilización de `first_value()`

Supongamos que tenemos que calcular para cada alumno la cantidad de libros máxima de entre todos los alumnos de cada ciudad, y calcular la diferencia entre ese valor y el valor de cada uno de los alumnos. Esta consulta podríamos crearla de la siguiente manera:

```
SELECT
 ciudad,
 nombre_apellidos,
 num_libros,
 FIRST_VALUE(num_libros) OVER
  (PARTITION BY ciudad ORDER BY num_libros DESC
    RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS prim_valor,
 FIRST_VALUE(num_libros) OVER
  (PARTITION BY ciudad ORDER BY num_libros DESC
    RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) - num_libros AS diff
FROM
  alumno
ORDER BY
  ciudad ASC,
  num_libros DESC
```

Los resultados que obtenemos son los siguientes: en este caso, la columna *Primer Valor* calcula el número de libros máximo para Barcelona (la columna *Ciudad* define la partición). Podemos ver cómo este valor se repite para cada alumno en Barcelona. En la segunda columna *Diferencia*, vemos que se calcula la diferencia entre la columna *Primer Valor* y el número de libros que dicho alumno tiene.

Ciudad	Nombre / Apellidos	Núm. Libros	Primer Valor	Diferencia
Barcelona	Alejandra Martínez	11	11	0
Barcelona	Manuel Vázquez	10	11	1
Barcelona	Elena Rodríguez	6	11	5
Barcelona	José Pérez	4	11	7
Lleida	Víctor Anllada	7	7	0
Lleida	Wenceslao Fernández	6	7	1
Lleida	Victoria Setan	5	7	2
Lleida	Felisa Sánchez	5	7	2
Lleida	José María Llopis	5	7	2
Tarragona	Marina Rodríguez	9	9	0
Tarragona	Fernando Nadal	8	9	1
Tarragona	Victoria Suarez	6	9	3

Fijaos que, al contrario que los demás ejemplos propuestos hasta este momento, hemos especificado en la consulta la cláusula `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`, que define el marco de la partición para cada fila actual (`CURRENT ROW`). Aunque el resultado de la consulta para `FIRST_VALUE` sería el mismo si en este caso no se especificara (ya que el marco estaría definido por `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`, que es el valor por defecto en el caso de omisión para PostgreSQL), es muy importante destacar que en otros casos, si se omite, nos proporcionaría unos resultados incorrectos.

3.4.7. *Last value*

Esta función devuelve el valor de una expresión asociado a la última fila del marco definido en la consulta. Esta función se define como `last_value()`, y contiene el siguiente parámetro:

```
last_value ( expression );
```

- `expression`: cualquier valor a evaluar excepto funciones analíticas (por ejemplo, una columna de una tabla, una función escalar, etc.).

Ejemplo de utilización de `last_value()`

Supongamos que tenemos un requisito similar al ejemplo propuesto para `first_value()`, esto es, calcular para cada alumno la cantidad de libros mínima de entre todos los alumnos de cada ciudad, y calcular la diferencia entre ese valor y el valor de cada uno de los alumnos. En lugar de calcular la cantidad de libros máxima, se requiere calcular la cantidad de libros **mínima**. Esta consulta podríamos crearla de la siguiente manera:

```
SELECT
  ciudad,
  nombre_apellidos,
  num_libros,
  LAST_VALUE(num_libros) OVER
    (PARTITION BY ciudad ORDER BY num_libros DESC
     RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS ult_valor,
  num_libros -
   LAST_VALUE(num_libros) OVER
     (PARTITION BY ciudad ORDER BY num_libros DESC
      RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS diff
FROM
  alumno
ORDER BY
  ciudad ASC,
  num_libros DESC
```

Los resultados que obtenemos son los siguientes:

Ciudad	Nombre / Apellidos	Núm. Libros	Último Valor	Diferencia
Barcelona	Alejandra Martínez	11	4	7
Barcelona	Manuel Vázquez	10	4	6
Barcelona	Elena Rodríguez	6	4	2
Barcelona	José Pérez	4	4	0
Lleida	Víctor Anllada	7	5	2
Lleida	Wenceslao Fernández	6	5	1
Lleida	Victoria Setan	5	5	0
Lleida	Felisa Sánchez	5	5	0
Lleida	José María Llopis	5	5	0
Tarragona	Marina Rodríguez	9	6	3
Tarragona	Fernando Nadal	8	6	2
Tarragona	Victoria Suarez	6	6	0

En esta función es muy importante especificar la cláusula de marco. La razón principal es porque PostgreSQL establece un marco por defecto `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`. Si no se especificase esta cláusula, el valor de *Último Valor* sería el **valor mínimo encontrado desde el inicio de la partición hasta la fila actual**. Veamos cómo serían los resultados en el caso de que no se especificase:

```
SELECT
  ciudad,
  nombre_apellidos,
  num_libros,
  LAST_VALUE(num_libros) OVER
    (PARTITION BY ciudad ORDER BY num_libros DESC) AS ult_valor,
  num_libros - LAST_VALUE(num_libros) OVER
     (PARTITION BY ciudad ORDER BY num_libros DESC) AS diff
FROM
  alumno
ORDER BY
  ciudad ASC,
  num_libros DESC
```

Los resultados que obtenemos a partir de esta consulta sin definición de marco no son los que deseábamos. Ved que *Último Valor* tiene el valor mínimo encontrado para cada ciudad **desde el inicio de la partición hasta la fila actual**, y **no hasta el fin de la partición**.

Ciudad	Nombre / Apellidos	Núm. Libros	Último Valor	Diferencia
Barcelona	Alejandra Martínez	11	11	0

Ciudad	Nombre / Apellidos	Núm. Libros	Último Valor	Diferencia
Barcelona	Manuel Vázquez	10	10	0
Barcelona	Elena Rodríguez	6	6	0
Barcelona	José Pérez	4	4	0
Lleida	Víctor Anllada	7	7	0
Lleida	Wenceslao Fernández	6	6	0
Lleida	Victoria Setan	5	5	0
Lleida	Felisa Sánchez	5	5	0
Lleida	José María Llopis	5	5	0
Tarragona	Marina Rodríguez	9	9	0
Tarragona	Fernando Nadal	8	8	0
Tarragona	Victoria Suarez	6	6	0

3.5. Uso de funciones de agregación como funciones analíticas

Además de las funciones analíticas explicadas anteriormente, PostgreSQL permite la utilización de funciones de agregación (como `SUM`, `AVG` o `COUNT`, entre otras) como funciones analíticas, tal y como se ha visto en alguno de los ejemplos vistos. En estos casos, la función se encarga de agregar las filas dentro del marco definido en la consulta.

Mediante el uso de funciones de agregación como funciones analíticas podemos solucionar problemas con alta complejidad de cálculo de manera más simple. Por ejemplo, podemos proporcionar solución a la necesidad de realizar **sumas acumulativas**. Veamos un ejemplo para clarificar este caso.

Ejemplo de suma acumulativa utilizando `SUM` como función analítica

Supongamos que queremos obtener un listado de nombres de alumnos, el número de asignaturas en las que se han matriculado, y la suma acumulativa de las asignaturas de cada alumno, ordenada por nombre de alumno. Esta consulta se crearía de la siguiente manera:

```
SELECT
  nombre_apellidos,
  num_asignaturas,
  SUM(num_asignaturas) OVER (ORDER BY nombre_apellidos ASC
     RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS suma_acumulativa
FROM
  alumno
ORDER BY
  nombre_apellidos ASC
```

Los resultados que obtenemos son los siguientes. Ved que lo que se obtiene es la suma de los elementos dentro del marco definido, que engloba desde la primera fila de la partición (que es toda la tabla) hasta la fila actual que se procesa.

Nombre / Apellidos	Núm. Asignaturas	Suma Acumulativa
Alejandra Martínez	4	4
Elena Rodríguez	3	7
Felisa Sánchez	2	9
Fernando Nadal	5	14
José María Llopis	4	18
José Pérez	9	27
Manuel Vázquez	7	34
Marina Rodríguez	4	38
Víctor Anllada	7	45
Victoria Setan	2	47
Victoria Suarez	8	55
Wenceslao Fernández	1	56

En estos casos, tenemos que tener cuidado de definir el marco de forma correcta. Si el marco se omite, por defecto se asume un marco `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`, que para el caso que hemos visto sería lo mismo. En cambio, si el marco se define diferente, podríamos obtener resultados que no son los esperados. La siguiente consulta realiza la suma acumulativa de asignaturas, con la diferencia de que el marco se ha definido desde el inicio hasta el fin de la partición (que es toda la tabla).

```
SELECT
  nombre_apellidos,
  num_asignaturas,
  SUM(num_asignaturas) OVER (ORDER BY nombre_apellidos ASC
     RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS suma_acumulativa
FROM
  alumno
ORDER BY
  nombre_apellidos ASC
```

Los resultados son totalmente diferentes de los que se esperaban, ya que se calcula la suma de todas las asignaturas para toda la partición de cada uno de los alumnos:

Nombre / Apellidos	Núm. Asignaturas	Suma Acumulativa
Alejandra Martínez	4	56
Elena Rodríguez	3	56
Felisa Sánchez	2	56
Fernando Nadal	5	56
José María Llopis	4	56
José Pérez	9	56
Manuel Vázquez	7	56
Marina Rodríguez	4	56
Víctor Anllada	7	56

Nombre / Apellidos	Núm. Asignaturas	Suma Acumulativa
Victoria Setan	2	56
Victoria Suarez	8	56
Wenceslao Fernández	1	56

4. Tratamiento de valores nulos

Un valor nulo (representado en SQL por el marcador `NULL`) es la forma que los SGBD relacionales utilizan para indicar que no existe información en una columna de una fila de una tabla. En otras palabras, es una manera de representar la falta de información que bien no sea aplicable o bien porque es desconocida. Un valor nulo, por lo tanto, no tiene ningún tipo de datos asociado: no es un dato numérico, ni una cadena de caracteres ni tampoco es un tipo de dato fecha.

Ejemplo de valor nulo

Supongamos que tenemos una tabla de empleados en una base de datos operacional de RRHH con la siguiente estructura: código, nombre, número de teléfono de la empresa y correo electrónico de la empresa, como se puede ver en la tabla inferior.

Empleados			
Código Empleado	**Nombre Empleado**	**Teléfono**	**Email**
U0001	Alejandra Martínez	652055678	amartinez@uoc.edu
U0002	José María Llopis	NULL	jmllopis@uoc.edu
U0003	Victoria Suarez	650091123	vsuarez@uod.edu
U0004	Victoria Setan	655112345	vsetan@uoc.edu
U0005	Víctor Anllada	NULL	NULL

Para los empleados José María Llopis y Víctor Anllada vemos que el teléfono tiene un valor nulo. La razón por la que José María Llopis no tiene un valor para teléfono es porque lo desconocemos, es decir, este usuario tiene un número de teléfono asignado por la empresa (se le ha visto en el departamento hablando por teléfono), pero desconocemos cuál es ya que no se ha proporcionado dicha información al departamento de RRHH. En el caso de Víctor Anllada, sabemos que este usuario no tiene número de teléfono (confirmado por el propio usuario), debido a que sus funciones en la empresa no requieren que este usuario tenga teléfono asignado (no aplicable) ni tampoco correo electrónico (no aplicable, también aparece con un valor nulo).

4.1. Valores nulos en BD operacionales

En las BD operacionales, los valores nulos nos permiten codificar la falta de información en tablas, como ya se ha comentado, bien porque no es aplicable o bien porque esta es desconocida. El uso de valores nulos en este tipo de BD puede causarnos dos tipos de problemas:

1) **Problemas de eficiencia**: este caso podría suceder cuando tenemos que acceder a filas de una tabla que tienen columnas con valores nulos. En estos casos, podría ser necesario revisar el modelo para generar una versión alternativa de la tabla, dependiendo de la proporción de datos (en relación con el total) que tengan valores nulos.

Ejemplo de consulta con posibles problemas de eficiencia

Supongamos que a la tabla de empleados mostrada en la introducción de esta sección se le añade una columna *Tasa Seguro*. Sabemos que los seguros privados se aplican solamente a empleados especiales (directivos), y de entre todos los empleados de la empresa solamente un grupo muy reducido de empleados disfrutan de este privilegio. Si queremos obtener aquellos empleados con seguro privado al que la empresa paga una tasa anual de seguro de más de 100 euros, la consulta sería la siguiente:

```
SELECT * FROM Empleados WHERE tasa_seguro >= 100
```

La consulta deberá acceder a todos los empleados (que se podría suponer de varios cientos) y verificar en cada uno de ellos si cumplen la condición para simplemente devolver, al final de la ejecución, un grupo de empleados muy reducido. En estos casos, se podría diseñar un modelo relacional en el que separamos los empleados por un lado y los empleados con seguro por otro lado, mejorando así la eficiencia de nuestra consulta.

2) Problemas de construcción de consultas: el hecho de que las consultas puedan involucrar atributos con valores nulos, nos obliga a que, a la hora de construir la consulta, tengamos que prestar atención a que el resultado que nos devuelva sea el correcto, tanto en el caso de que existan valores nulos o no.

Ejemplo de consulta con tratamiento de nulos

Suponiendo la tabla de empleados mostrada en la introducción de esta sección, y una segunda tabla con información sobre usuarios de red mostrada a continuación.

Usuarios		
Usuario	**Acceso a Campus**	**Acceso a Biblioteca**
amartinez@uoc.edu	Si	No
vsuarez@uod.edu	No	Si
vsetan@uoc.edu	Si	Si

Queremos obtener aquellos empleados que no son usuarios. Las consultas propuestas son:

```
SELECT * FROM Empleados
WHERE email NOT IN (SELECT usuario FROM Usuarios);

SELECT * FROM Empleados e
WHERE NOT EXISTS
        (SELECT * FROM Usuarios u WHERE e.email = u.usuario);
```

La primera consulta nos devolverá el usuario con código de empleado U0002, mientras que la segunda consulta nos devolverá los usuarios con código de empleado U0002 y U0005. La segunda consulta es la que nos proporciona los resultados correctos.

Además de lo comentado, es importante destacar que el hecho de considerar valores nulos en un SGBD hace que podamos decir que dejamos de trabajar con una **lógica binaria o bivalente** (verdadero/falso) y comenzamos a trabajar con una **lógica ternaria o trivalente** (verdadero/falso/desconocido), que requiere de un tratamiento especial para identificar aquellos valores que son

desconocidos o NULL (en SQL se realiza mediante las cláusulas IS NULL/IS NOT NULL). En las siguientes tablas podemos ver cómo se evaluarían las operaciones lógicas AND, OR y NOT en lógica ternaria a diferencia de la lógica binaria.

Lógica Ternaria				
A	**B**	**A OR B**	**A AND B**	**NOT A**
Verdadero	**Verdadero**	Verdadero	Verdadero	Falso
Verdadero	**NULL**	Verdadero	NULL	Falso
Verdadero	**Falso**	Verdadero	Falso	Falso
NULL	**Verdadero**	Verdadero	NULL	NULL
NULL	**NULL**	NULL	NULL	NULL
NULL	**Falso**	NULL	Falso	NULL
Falso	**Verdadero**	Verdadero	Falso	Verdadero
Falso	**NULL**	NULL	Falso	Verdadero
Falso	**Falso**	Falso	Falso	Verdadero

Lógica Binaria				
A	**B**	**A OR B**	**A AND B**	**NOT A**
Verdadero	**Verdadero**	Verdadero	Verdadero	Falso
Verdadero	**Falso**	Verdadero	Falso	Falso
Falso	**Verdadero**	Verdadero	Falso	Verdadero
Falso	**Falso**	Falso	Falso	Verdadero

4.2. Valores nulos en almacenes de datos

Al igual que lo visto anteriormente en las BD operacionales, es posible utilizar los valores nulos en los almacenes de datos para codificar la falta de información en las tablas. En cambio, dada la naturaleza de estos sistemas, la problemática del uso de valores nulos en este tipo de BD es ligeramente diferente a la expuesta en las BD operacionales. A continuación, vamos a ver cómo podemos realizar el tratamiento de nulos desde dos puntos de vista: tratamiento de nulos en tablas dimensiones y tratamiento de nulos en tablas de hechos.

4.2.1. Valores nulos en tablas de dimensiones

Las dimensiones, en el contexto de los almacenes de datos, representan el punto de vista que se utiliza en el análisis de los datos. Suelen almacenar **información descriptiva** o **cualitativa**, como nombres, códigos o descripciones, y se utilizan para darle un contexto a la información **cuantitativa** (métricas), la cual procede de las tablas de hechos.

Es muy común encontrarse casos en los que ciertas columnas que pertenecen a dimensiones no tienen datos, bien porque estos no existen en las BD operacionales, o simplemente porque la columna no es aplicable para la fila en cuestión.

Ejemplo de dimensión y datos desconocidos/no aplicables

Un ejemplo de dimensión en almacenes de datos es Cliente. Para el cliente, se suele guardar el identificador del cliente, el nombre, la dirección, el código postal y la ciudad donde reside. Generalmente, estos datos suelen existir en los sistemas operacionales de una empresa, aunque no es inusual que el código postal no esté almacenado. Este es un ejemplo claro de dato que almacenará en origen un valor nulo.

También existen columnas que solamente tienen sentido dependiendo de las características de la fila. Por ejemplo, si se trata de un cliente individual, el sistema operacional podría almacenar información de estado civil. Este tipo de columna no tiene sentido si se trata de una corporación, por lo que en este caso se almacenaría en origen un valor nulo.

Aunque es común la utilización de valores nulos en BD operacionales, estos no suelen ser bien recibidos por parte de los diseñadores de almacenes de datos. Las razones principales son las siguientes:

a) Diferentes SGBD tienen diferentes comportamientos a la hora de tratar valores nulos en cláusulas `WHERE`, agrupaciones (`GROUP BY`) y restricciones de integridad.

b) Del mismo modo, diferentes herramientas de creación de informes pueden tratar los nulos de diferentes maneras, sobre todo a la hora de combinar datos procedentes de diferentes consultas a través de información conformada.

c) Consultas multihecho (consultas que recuperan datos desde diferentes tablas de hecho utilizando dimensiones conformadas): la forma en la que estas consultas se realizan depende de las herramientas de creación de informes. Un mecanismo muy utilizado es la utilización de `FULL OUTER JOIN`, lo que permite combinar varias subconsultas que «atacan» a una tabla de hechos concreta a partir de datos conformados. Asegurando valores por defecto, garantizamos la consistencia de los datos a la hora de mostrarlos en un informe.

FULL OUTER JOIN

Combinación externa (*outer join*) que nos permite obtener todos los valores de ambas tablas.

d) Evitar valores nulos en listas de valores de informes.

e) Existen productos OLAP en el mercado que no aceptan valores nulos, lo que nos causaría problemas a la hora de generar cubos de datos.

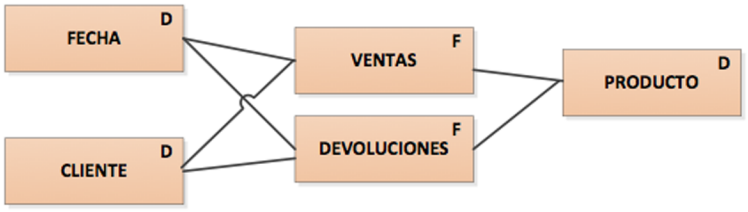
OLAP

On line analytical processing

Ejemplo de consulta multihecho con `FULL OUTER JOIN`

Supongamos que tenemos un modelo en estrella para análisis de una empresa con Fecha, Cliente y Producto como dimensiones (D), y Ventas y Devoluciones como tablas de hechos (F), tal y como se muestra en la figura 3.

Figura 3. Diseño en estrella propuesto



Los datos de cada una de estas tablas es el siguiente. Las claves primarias son claves subrogadas.

Fecha			
Id Fecha	**Fecha**	**Año / Mes**	**Año**
20150101	01-01-2015	2015/01	2015
20150102	02-01-2015	2015/01	2015
20150103	03-01-2016	2015/01	2015
20150104	04-01-2016	2015/01	2015

Cliente					
Id Cliente	**Código Cliente**	**Nombre Cliente**	**Dirección**	**Código Postal**	**Ciudad**
1	1111	USACO S.A.	C/ Alpargata 22	31-031	Barcelona
2	2222	PENTEX S.A.	C/ Mediana 22	31-067	Tarragona
3	3333	SEMITEX S.A.	C/ Superior 102	NULL	Lleida
4	4444	FEDEX S.A.	C/ Inferior 55	30-070	Barcelona

Véase que el código postal del cliente con clave subrogada 3 (SEMITEX S.A.) es nulo.

Producto			
Id Producto	**Código Producto**	**Nombre Producto**	**Precio Actual**
1	P0001	GALLETAS SENSACION 100gr	2.45
2	P0002	BARRA PAN 250gr	1
3	P0003	MANZANAS GALA 1Kg	1.5
4	P0004	NARANJAS VALENCIA 1Kg	1.25

Producto			
Id Producto	**Código Producto**	**Nombre Producto**	**Precio Actual**
5	P0005	PLATANOS CANARIAS 1Kg	0.98

Ventas					
Id Ventas	**Fecha**	**Cliente**	**Producto**	**No.Productos**	**Ventas (EUR)**
1	20150101	1	1	10	24.5
2	20150101	1	2	5	5
3	20150101	2	3	3	4.5
4	20150101	2	4	1	1.25
5	20150101	3	5	NULL	14.7

Véase que el valor de No. Productos para la venta del cliente 3 (última fila) es nulo. Los valores de Fecha, Cliente y Producto referencian a las claves subrogadas de las tablas de Fecha, Cliente y Producto respectivamente.

Devoluciones					
Id Devolución	**Fecha**	**Cliente**	**Producto**	**No.Productos**	**Devuelto (EUR)**
1	20150103	1	1	5	12.25
2	20150103	2	2	1	1
3	20150103	3	3	1	1.5

Los valores de Fecha, Cliente y Producto referencian a las claves subrogadas de las tablas de Fecha, Cliente y Producto respectivamente.

Imaginemos que nos piden generar la siguiente consulta: obtener el año/mes, nombre de cliente, código postal, ventas (en euros) y devoluciones (en euros) ordenados por nombre de cliente ascendentemente.

Como podéis imaginar, se trata de una consulta multihecho: por un lado, debemos obtener la fecha, el nombre del cliente y las ventas (usando la tabla de hechos de ventas) y, por otro lado, debemos obtener la fecha, el nombre de cliente y las devoluciones (usando la tabla de hechos de devoluciones). Para facilitar el ejemplo, mostramos los resultados que esperamos obtener:

Año / Mes	Nombre Cliente	Código Postal	Ventas (EUR)	Devuelto (EUR)
2015/01	PENTEX S.A.	31-067	5.75	1
2015/01	SEMITEX S.A.	NULL	14.7	1.5
2015/01	USACO S.A.	31-031	29.5	12.25

Para obtener los datos de ventas realizaremos la siguiente consulta:

```
SELECT
  fecha.ano_mes,
  cliente.nombre_cliente,
  cliente.cod_postal,
  SUM(ventas.ventas_eur) AS ventas_eur
FROM
  ventas
INNER JOIN fecha
ON
  ventas.fecha_skey = fecha.fecha_skey
INNER JOIN cliente
ON
  ventas.cliente_skey = cliente.cliente_skey
GROUP BY
  fecha.ano_mes,
  cliente.nombre_cliente,
  cliente.cod_postal
ORDER BY
  cliente.nombre_cliente
```

Año / Mes	Nombre Cliente	Código Postal	Ventas (EUR)
2015/01	PENTEX S.A.	31-067	5.75
2015/01	SEMITEX S.A.	NULL	14.7
2015/01	USACO S.A.	31-031	29.5

Por otro lado, para obtener los datos de devoluciones realizaremos la siguiente consulta:

```
SELECT
  fecha.ano_mes,
  cliente.nombre_cliente,
  cliente.cod_postal,
  SUM(devoluciones.devuelto_eur) AS devuelto_eur
FROM
  devoluciones
INNER JOIN fecha
ON
  devoluciones.fecha_skey = fecha.fecha_skey
INNER JOIN cliente
ON
  devoluciones.cliente_skey = cliente.cliente_skey
GROUP BY
  fecha.ano_mes,
  cliente.nombre_cliente,
  cliente.cod_postal
ORDER BY
  cliente.nombre_cliente
```

Año / Mes	Nombre Cliente	Código Postal	Devuelto (EUR)
2015/01	PENTEX S.A.	31-067	1
2015/01	SEMITEX S.A.	NULL	1.5
2015/01	USACO S.A.	31-031	12.25

El último paso es el de combinar ambas consultas en una sola. La propuesta de consulta, utilizando `FULL OUTER JOIN`, se puede ver a continuación:

```
SELECT
      COALESCE(consulta_1.ano_mes, consulta_2.ano_mes) AS ano_mes,
      COALESCE(consulta_1.nombre_cliente, consulta_2.nombre_cliente) AS nombre_cliente,
      COALESCE(consulta_1.cod_postal, consulta_2.cod_postal) AS cod_postal,
      consulta_1.ventas_eur,
      consulta_2.devuelto_eur
FROM
  (
    SELECT
      fecha.ano_mes,
      cliente.nombre_cliente,
      cliente.cod_postal,
      SUM(ventas.ventas_eur) AS ventas_eur
    FROM
      ventas
    INNER JOIN fecha
      ON ventas.fecha_skey = fecha.fecha_skey
    INNER JOIN cliente
      ON ventas.cliente_skey = cliente.cliente_skey
    GROUP BY
      fecha.ano_mes,
      cliente.nombre_cliente,
      cliente.cod_postal
  )
  consulta_1
FULL OUTER JOIN
  (
    SELECT
      fecha.ano_mes,
      cliente.nombre_cliente,
      cliente.cod_postal,
      SUM(devoluciones.devuelto_eur) AS devuelto_eur
    FROM
      devoluciones
    INNER JOIN fecha
      ON devoluciones.fecha_skey = fecha.fecha_skey
    INNER JOIN cliente
      ON devoluciones.cliente_skey = cliente.cliente_skey
    GROUP BY
      fecha.ano_mes,
      cliente.nombre_cliente,
      cliente.cod_postal
  )
  consulta_2 ON
         consulta_1.ano_mes = consulta_2.ano_mes
     AND consulta_1.nombre_cliente = consulta_2.nombre_cliente
     AND consulta_1.cod_postal = consulta_2.cod_postal
ORDER BY
  nombre_cliente ASC
```

Si ejecutamos esta consulta, los datos obtenidos serían los siguientes, que difieren ligeramente de los datos esperados, en concreto para el cliente SEMITEX S. A.

Año / Mes	Nombre Cliente	Código Postal	Ventas (EUR)	Devuelto (EUR)
2015/01	PENTEX S.A.	31-067	5.75	1
2015/01	SEMITEX S.A.	NULL	NULL	1.5
2015/01	SEMITEX S.A.	NULL	14.7	NULL
2015/01	USACO S.A.	31-031	29.5	12.25

¿Por qué sucede esto? Vemos que en la consulta multihecho, una de las condiciones del `FULL OUTER JOIN` es `consulta_1.cod_postal = consulta_2.cod_postal`. En este caso, el código postal es nulo para dicho cliente, por lo que la condición de igualdad no se cumple. El resultado de los datos para este cliente aparece fracturado en dos filas: una con los datos de ventas y otra con los datos de devoluciones. Véase también que, para los datos de ventas, el valor de devoluciones es `NULL`, y viceversa, el valor de ventas en devoluciones es `NULL`. Esto es así porque no existe información disponible para dicha columna desde la tabla de hechos a la que referencia.

Nota

`COALESCE` es una función que devuelve el primer valor no nulo de la lista de valores proporcionada.

Si dispusiésemos de valores por defecto, los resultados obtenidos serían diferentes. Asumiendo un valor por defecto *Desconocido* para el código postal, los resultados obtenidos utilizando esta misma consulta serían los siguientes:

Año / Mes	Nombre Cliente	Código Postal	Ventas (EUR)	Devuelto (EUR)
2015/01	PENTEX S.A.	31-067	5.75	1
2015/01	SEMITEX S.A.	DESCONOCIDO	14.7	1.5
2015/01	USACO S.A.	31-031	29.5	12.25

A partir de lo comentado anteriormente, la metodología de diseño de modelos dimensionales recomienda que **todas las columnas de las dimensiones tengan un valor por defecto**. Esto es:

- Para columnas con tipo de dato cadena de caracteres, se suele asignar un valor *Desconocido* o *No Aplicable* (*N/A*), dependiendo de su naturaleza.
- Para columnas con tipo de dato numérico, se suele utilizar un valor 0.
- Para columnas con tipo de dato fecha/hora, se suele utilizar una fecha lejana o temprana en el tiempo. Por ejemplo, 1900-01-01 o 9999-12-31, que suelen indicar «desde el inicio de la historia» y «hasta el fin de la historia» respectivamente.

Nota

La metodología de diseño de modelos dimensionales es la propuesta por Ralph Kimball mediante su obra titulada *The Data Warehouse Toolkit*, referenciada en la bibliografía de este módulo.

Es importante destacar que los valores propuestos anteriormente son orientativos y que estos suelen acordarse con los usuarios a la hora de diseñar el almacén de datos.

4.2.2. Valores nulos en tablas de hechos

Al contrario que las dimensiones, las tablas de hechos contienen la información que queremos medir, información que denominamos **cuantitativa**. Esta información es la que agregamos mediante las funciones de agregación (`SUM`, `AVG`, `COUNT`...), información que combinamos con dimensiones para obtener los diferentes puntos de vista que buscamos.

Las medidas, que generalmente suelen ser de tipo numérico, podrían no disponer de un valor concreto (bien por motivos de calidad de datos, o simplemente porque no era necesario guardarlo), siendo este un escenario bastante frecuente. En estos casos, al ser valores que van a ser agregados, no tendríamos ningún problema en permitir valores nulos. La razón por la que esto es posible es por la naturaleza de estas funciones, que no consideran los nulos como un valor a considerar. Por ejemplo, la suma (`SUM`) de valores nulos y no nulos equivale a sumar solamente aquellos valores que no son nulos. Lo mismo sucede con la función media (`AVG`), contar (`COUNT`), máximo (`MAX`) y mínimo (`MIN`).

Ejemplo de agregación de valores nulos

Utilizando el ejemplo anterior de ventas, podemos ver que el valor de No. Productos en la venta realizada para el cliente con clave subrogada 3 (SEMITEX S. A.) tiene un valor

nulo. Si calculamos la **suma**, **media**, **máximo**, **mínimo**, y el **conteo** de esta medida para año/mes y cliente, obtendremos los siguientes resultados.

Año / Mes	Nombre Cliente	Suma Núm. Prod.	Avg Núm. Prod.	Max Núm. Prod.	Min Núm. Prod.	Count Núm.Ventas
2015/01	PENTEX S.A.	4	2	3	1	2
2015/01	SEMITEX S.A.	NULL	NULL	NULL	NULL	0
2015/01	USACO S.A.	15	7.5	10	5	2

Podemos ver que todas las funciones de agregación no tienen en cuenta los valores nulos a la hora de realizar los cálculos. Si hubiésemos puesto un valor por defecto (por ejemplo, 0), entonces obtendríamos los siguientes resultados. Ved que para el conteo, el valor que aparece ahora es un 1, ya que existe un valor (que es el 0 por defecto que hemos puesto).

Año / Mes	Nombre Cliente	Suma Núm. Prod.	Avg Núm. Prod.	Max Núm. Prod.	Min Núm. Prod.	Count Núm.Ventas
2015/01	PENTEX S.A.	4	2	3	1	2
2015/01	SEMITEX S.A.	0	0	0	0	1
2015/01	USACO S.A.	15	7.5	10	5	2

En lo que respecta a métricas, es muy importante entender qué función de agregación se va a utilizar para poder decidir si es necesario utilizar valores por defecto o no, y cuál es el valor más adecuado.

Por otro lado, las tablas de hechos contienen referencias a las dimensiones. Estas referencias son las claves subrogadas que hemos estudiado anteriormente, y que identifican unívocamente a las filas de la dimensión. En ocasiones, es posible encontrarse casos en los que, a la hora de mapear el valor para obtener la clave subrogada de la dimensión para insertarla en la tabla de hechos, no es posible encontrar dicho valor (por ejemplo, si este nunca ha existido en el sistema operacional). En estos casos, al no obtener una clave subrogada, el valor a insertar sería un nulo.

Cuando nos encontramos en este escenario, la recomendación es utilizar un valor de clave subrogada especial para evitar tener valores nulos. Si recordamos lo visto en la sección de claves subrogadas, habíamos hablado de una fila especial con clave subrogada -1, que nos permitía identificar la falta de valores en los sistemas origen. Así, garantizaremos que todas las filas de la tabla de hechos estén vinculadas siempre a alguna fila existente en la dimensión.

Ejemplo de fila especial para capturar falta de valores

Utilizando la dimensión Cliente como ejemplo, la fila especial con clave subrogada -1 se podría implementar de la siguiente forma. Así, conseguiríamos mapear correctamente las ventas y devoluciones asociadas a clientes que no podemos encontrar en la dimensión.

Cliente					
Id Cliente	**Código Cliente**	**Nombre Cliente**	**Dirección**	**Código Postal**	**Ciudad**
-1	DESCONOCIDO	DESCONOCIDO	DESCONOCIDO	DESCONOCIDO	DESCONOCIDO
1	1111	USACO S.A.	C/ Alpargata 22	31-031	Barcelona
2	2222	PENTEX S.A.	C/ Mediana 22	31-067	Tarragona
3	3333	SEMITEX S.A.	C/ Superior 102	NULL	Lleida
4	4444	FEDEX S.A.	C/ Inferior 55	30-070	Barcelona

Existen varias razones para utilizar este mecanismo:

- Evitar violar la integridad referencial entre dimensiones y hechos.
- Facilitar el uso de `INNER JOIN` en lugar de `LEFT OUTER JOIN`, mejorando el rendimiento de la consulta.
- Asegurarse que los resultados agregados de la consulta son siempre correctos, independientemente de la dimensión que se utilice en el análisis, al estar todos los datos correctamente mapeados.
- Facilitar la identificación de problemas de calidad de datos. Al realizar informes y detectar valores *DESCONOCIDO* o similares puede ser una señal de que existen problemas en el sistema operacional.

De hecho, la recomendación es diseñar la integridad referencial entre dimensiones y hechos con columnas que no permitan nulos (`NOT NULL`), forzando la utilización de estas filas especiales en los casos descritos anteriormente.

5. Transacciones

Uno de los objetivos más importantes de los SGBD es garantizar la integridad de los datos almacenados en las BD que gestionan.

La integridad tiene que ver con la consistencia y la calidad de los datos. Hay diversas causas que pueden comprometer esa integridad: el acceso simultáneo de usuarios diferentes a una misma BD, una situación de avería, el hecho de que se haya decidido tener datos replicados para mejorar el rendimiento en el acceso a la BD o que una operación pueda comprometer una regla de integridad definida sobre la BD.

En este apartado veremos las posibles anomalías que se derivan del acceso simultáneo de diversos usuarios a la misma BD y en el hecho de asegurar la disponibilidad de la BD ante fallos o desastres, como sería el caso de una avería en los dispositivos de almacenamiento externo, un apagón o un incendio.

Hay que tener presente que los datos de una organización casi siempre son uno de sus activos principales, una herramienta indispensable para el desarrollo normal de las actividades que lleva a cabo.

Por lo tanto, el SGBD tiene que afrontar todas estas posibles anomalías y, para hacerlo, se fundamenta en el concepto de transacción y en una serie de mecanismos para gestionar esas transacciones.

5.1. Problemática asociada a la gestión de transacciones

En los SGBD, el concepto de transacción representa la unidad de trabajo por lo que se refiere a control de concurrencia y recuperación. La gestión de transacciones que realiza el SGBD protege las aplicaciones de las anomalías importantes que se pueden producir si no se lleva a cabo.

A continuación veremos, con ejemplos, los problemas que pueden surgir cuando se ejecutan de forma concurrente, y sin ningún control por parte del SGBD, diferentes transacciones.

Supongamos que una aplicación de una entidad bancaria ofrece a los usuarios una función que permite transferir cierta cantidad de dinero de una cuenta de origen a una cuenta de destino.

Esta función podría ejecutar los pasos que mostramos en la siguiente tabla (en pseudocódigo):

Transferencia de cantidad Q de cuenta_origen a cuenta_destino	
N.° operación	**Operaciones que hay que ejecutar**
1	saldo_origen := leer_saldo(cuenta_origen) Comprobar que saldo_origen es mayor o igual a Q (suponemos que hay saldo suficiente para hacer la transferencia)
2	saldo_destino := leer_saldo(cuenta_destino)
3	escribir_saldo(cuenta_origen, saldo_origen – Q)
4	escribir_saldo(cuenta_destino, saldo_destino + Q)
5	registrar_movimiento("transferencia", cuenta_origen, cuenta_destino, Q) Crear un registro para anotar la transferencia en una tabla de movimientos, y poner también la fecha y la hora, por ejemplo

Hay que considerar las anomalías que se producirán si no se toma ninguna precaución:

1) Supongamos que un usuario empieza a ejecutar una de estas transferencias y, justo después del tercer paso, un apagón hace que el proceso no acabe. En este caso, se habrá restado la cantidad transferida al saldo de la cuenta de origen, pero no se habrá sumado al de la cuenta de destino. Esta posibilidad representa un peligro grave.

Desde el punto de vista de la aplicación, las operaciones que se ejecutan cuando se hace la transferencia se tienen que llevar a cabo completamente o no se tienen que efectuar en absoluto; es decir, la transferencia no puede quedar a medias.

2) Supongamos que dos usuarios diferentes (A y B) intentan hacer dos transferencias al mismo tiempo desde cuentas de origen diferentes y hacia la misma cuenta de destino. Analicemos qué puede pasar si, por cualquier motivo y sin ningún control por parte del SGBD, los pasos de las transacciones se ejecutan de forma concurrente en el siguiente orden:

Ejecución concurrente de dos transferencias bancarias			
N.° operación	**Transferencia usuario A (Q = 20)**	**N.° operación**	**Transferencia usuario B (Q = 40)**
1	saldo_origen:= leer_saldo(cuenta_origen1) Comprobar que saldo_origen es mayor o igual a 20 (suponemos que hay saldo suficiente para hacer la transferencia)		
2	saldo_destino:= leer_saldo(cuenta_destino)		
		1	saldo_origen:= leer_saldo(cuenta_origen2) Comprobar que saldo_origen es mayor o igual a 40 (suponemos que hay saldo suficiente para hacer la transferencia)

Ejecución concurrente de dos transferencias bancarias			
N.° operación	**Transferencia usuario A (Q = 20)**	**N.° operación**	**Transferencia usuario B (Q = 40)**
		2	saldo_destino:= leer_saldo(cuenta_destino)
3	escribir_saldo(cuenta_origen1, saldo_origen – 20)		
4	escribir_saldo(cuenta_destino, saldo_destino + 20)		
5	registrar_movimiento("transferencia", cuenta_origen1, cuenta_destino, 20)		
		3	escribir_saldo(cuenta_origen2, saldo_origen – 40)
		4	escribir_saldo(cuenta_destino, saldo_destino + 40)
		5	registrar_movimiento("transferencia", cuenta_origen2, cuenta_destino, 40)

El resultado final es que la cuenta de destino tiene como saldo la inicial más 40, en vez de más 60. Esto es incorrecto, ya que se ha perdido la cantidad que ha transferido el usuario A.

Hay que impedir de alguna manera que el acceso concurrente de diversos usuarios produzca resultados anómalos.

Cada usuario, individualmente, tiene que tener la percepción de que solo él trabaja con la BD. En el ejemplo que hemos planteado, la ejecución de la transferencia que efectúa el usuario B ha interferido en la ejecución de la transferencia que lleva a cabo el usuario A. Si las dos transferencias se hubieran ejecutado correctamente aisladas la una de la otra, el saldo total de la cuenta de destino habría sido el saldo inicial más 60.

3) Imaginemos que un error de programación de la función de transferencia hace que el saldo de la cuenta de destino reciba como nuevo valor la cantidad que se ha transferido, en vez de sumarla al saldo anterior. Naturalmente, este comportamiento será incorrecto, ya que no se corresponde con el deseo de los usuarios, y dejará la BD en un estado inconsistente: los saldos que tendrían que tener las cuentas de acuerdo con los movimientos registrados (en el quinto paso) no coincidirían con los que se han almacenado realmente.

En conclusión, es misión de los diseñadores y los programadores que las transacciones verifiquen los requisitos de los usuarios.

4) Planteémonos qué pasaría si, después de utilizar la aplicación durante unos cuantos días y en un momento de plena actividad, se produce un error fatal del dispositivo de almacenamiento externo en el que se guarda la BD, de manera que esta deja de estar disponible.

En definitiva, tiene que haber mecanismos para evitar la pérdida tanto de los datos más antiguos como de las actualizaciones más recientes.

5.2. Definición y propiedades de las transacciones

El acceso a los datos que hay en una BD se hace mediante la ejecución de operaciones en el SGBD correspondiente. Puesto que estamos interesados en SGBD relacionales, estas operaciones, a alto nivel, serán sentencias SQL. Además, con vistas a resolver el tipo de problemas que hemos planteado en el apartado anterior, estas operaciones se agrupan en transacciones.

Una **transacción** es un conjunto de operaciones (de lectura y actualización) sobre la BD que se ejecutan como una unidad indivisible de trabajo. La transacción acaba su ejecución confirmando o cancelando los cambios que se han llevado a cabo sobre la BD.

Un programa empieza a trabajar con una BD conectándose a ella de una manera adecuada y estableciendo una sesión de trabajo que permite efectuar operaciones de lectura y actualización (inserciones, borrados, modificaciones) de la BD. Para hacer una operación tiene que haber una transacción activa (o en ejecución), que siempre es única. La transacción activa se puede iniciar mediante una instrucción especial o automáticamente cuando se hace la primera operación en el SGBD.

Toda transacción debería cumplir cuatro propiedades, conocidas como propiedades **ACID**:

ACID

ACID es una sigla que se forma a partir de las iniciales de las palabras atomicidad, consistencia, aislamiento y definitividad (*atomicity*, *consistency*, *isolation* y *definitivity*).

1) Atomicidad. El conjunto de operaciones que constituyen la transacción es la unidad atómica, indivisible, de ejecución. Esto quiere decir que, o bien se ejecutan todas las operaciones de la transacción (y, en este caso, la transacción confirma los resultados), o bien no se ejecuta ninguna en absoluto (y, en este caso, la transacción cancela los resultados). En definitiva, el SGBD tiene que garantizar el todo o nada para cada transacción:

a) Para confirmar los resultados producidos por la ejecución de una transacción, disponemos de la sentencia SQL `COMMIT`.

b) En el caso contrario, ya sea porque alguna cosa impide que se acabe de ejecutar la transacción (por ejemplo, un corte de luz), ya sea porque la transacción acaba con una petición explícita de cancelación por parte del programa

de aplicación, el SGBD tiene que deshacer todos los cambios que la transacción haya hecho sobre la BD hasta ese momento, como si dicha transacción nunca hubiera existido. En los dos casos se dice que la transacción ha abortado (en inglés, *abort*) la ejecución. Para cancelar de manera explícita los resultados producidos por la ejecución de una transacción, disponemos de la sentencia SQL `ROLLBACK`.

2) Consistencia. La ejecución de una transacción tiene que preservar la consistencia de la BD. En otras palabras, si antes de ejecutarse una transacción la BD se encuentra en un estado consistente (es decir, en un estado en el que se verifican todas las reglas de integridad definidas sobre la BD), al acabar la ejecución de la transacción la BD también tiene que quedar en un estado consistente, si bien, mientras la transacción esté activa, la BD podría caer momentáneamente en un estado inconsistente.

3) Aislamiento. Una transacción no puede ver interferida su ejecución por ninguna otra transacción que se esté ejecutando de forma concurrente con esta. En definitiva, el SGBD tiene que garantizar el correcto aislamiento de las transacciones.

4) Definitividad. Los resultados producidos por una transacción que confirma (es decir, que ejecuta la operación de `COMMIT`) tienen que ser definitivos en la BD; nunca se pueden perder, independientemente de que se produzcan fallos o desastres, hasta que otra transacción cambie esos resultados y los confirme. Al contrario, los resultados producidos por una transacción que aborta su ejecución se han de descartar de la BD.

Es importante destacar que las propiedades que acabamos de presentar no son independientes entre ellas; por ejemplo, las propiedades de atomicidad y de definitividad están estrechamente interrelacionadas. Además, el hecho de garantizar las propiedades ACID de las transacciones no es solamente una misión del SGBD, sino también de las aplicaciones que lo utilizan y, por consiguiente, de su desarrollador.

5.3. Interferencias entre transacciones

En este apartado presentaremos los tipos de interferencias que se pueden producir si las transacciones que se ejecutan de manera concurrente no verifican la propiedad de aislamiento.

Antes de entrar en estas interferencias, es importante destacar que, si hay dos transacciones que se ejecutan de forma concurrente, una de estas sólo puede interferir en la ejecución de la otra si se dan las siguientes circunstancias:

a) Las dos transacciones acceden a una misma porción de la BD.

b) Como mínimo una de las dos transacciones, sobre esta porción común de la BD a la que acceden, efectúa operaciones de actualización.

En otras palabras, cuando las transacciones que se ejecutan de forma concurrente solo hacen lecturas, no se producirán nunca interferencias. De manera similar, en el caso de que las transacciones hagan actualizaciones, si estas se realizan sobre porciones diferentes, no relacionadas en la BD, tampoco se pueden producir interferencias.

A continuación presentamos, mediante ejemplos, los tipos de interferencias que puede haber entre dos transacciones T1 y T2 que se procesan de forma concurrente si no están aisladas de una manera adecuada entre ellas:

1) Actualización perdida. Esta interferencia se produce cuando se pierde un cambio efectuado por una transacción sobre un dato a causa de la presencia de otra transacción que también cambia el mismo dato. Esto podría suceder en una situación como la que se muestra a continuación:

Transacción T1 (reintegro de 20)	Transacción T2 (reintegro de 40)
saldo := leer_saldo(cuenta)	
	saldo := leer_saldo(cuenta)
escribir_saldo(cuenta, saldo – 20)	
	escribir_saldo(cuenta, saldo – 40)
COMMIT	
	COMMIT

T1 y T2 ejecutan un mismo tipo de transacción; en este caso, un reintegro de una misma cuenta bancaria. Las dos transacciones leen el mismo valor del saldo de la cuenta, lo actualizan de forma independiente (asumimos que hay saldo suficiente en la cuenta para hacer los reintegros) y restan a este saldo la cantidad que se ha sustraído.

Suponiendo que el SGBD ejecuta las operaciones que constituyen cada transacción sin ningún control y en el orden que se propone en el ejemplo, el cambio correspondiente a la sustracción de T1 se pierde. En consecuencia, el saldo disminuye sólo en 40, en vez de en 60.

En definitiva, T1 ha visto interferida su ejecución a causa de la presencia de T2. Si el orden de ejecución de las operaciones de cada transacción hubiera sido el siguiente:

Transacción T1 (reintegro de 20)	Transacción T2 (reintegro de 40)
saldo := leer_saldo(cuenta)	
	saldo := leer_saldo(cuenta)
	escribir_saldo(cuenta, saldo – 40)
escribir_saldo(cuenta, saldo – 20)	
COMMIT	
	COMMIT

se habría producido igualmente la interferencia. En este caso, se habría perdido el cambio efectuado por T2. En consecuencia, el saldo disminuiría solo en 20, en vez de en 60. En este caso, T2 habría visto interferida su ejecución a causa de la presencia de T1.

En definitiva, la interferencia ocurre porque se producen dos lecturas consecutivas de un mismo dato (el saldo de una misma cuenta) seguidas de dos cambios consecutivos del mismo dato (de nuevo, el saldo de una misma cuenta). Simplemente, si la secuencia de operaciones hubiera sido, por ejemplo, la que se muestra a continuación:

Transacción T1 (reintegro de 20)	Transacción T2 (reintegro de 40)
saldo := leer_saldo(cuenta)	
escribir_saldo(cuenta, saldo – 20)	
	saldo := leer_saldo(cuenta)
	escribir_saldo(cuenta, saldo – 40)
COMMIT	
	COMMIT

la interferencia no se habría producido. En este caso, T2 recuperaría el valor del saldo de cuenta dejado por T1 y, teniendo en cuenta este nuevo valor de saldo para la cuenta, efectuaría su propio reintegro. En consecuencia, el saldo de la cuenta disminuiría en 60.

2) Lectura no confirmada. Esta interferencia se puede producir cuando una transacción recupera un dato pendiente de confirmación que ha sido modificado por otra transacción que se ejecuta de forma concurrente con la transacción que recupera el dato. Eso podría suceder en diversas situaciones, como las que se muestran a continuación:

Transacción T1 (consulta saldo)	Transacción T2 (reintegro de 20)
	saldo := leer_saldo(cuenta)

Transacción T1 (consulta saldo)	Transacción T2 (reintegro de 20)
	escribir_saldo(cuenta, saldo – 20)
saldo := leer_saldo(cuenta)	
COMMIT	
	ROLLBACK

Primeramente, la transacción T2 lee el saldo de la cuenta y lo disminuye en la cantidad que se quiere reintegrar. A continuación, la transacción T1 efectúa una consulta de saldo de la misma cuenta sobre la que T2 hace el reintegro. El valor de saldo que obtiene T1 está pendiente de confirmar, es un dato provisional, ya que T2 todavía no ha confirmado sus resultados. Acto seguido, la transacción T1 finaliza su ejecución y confirma los resultados. Finalmente, T2 cancela su ejecución. Esta cancelación causa que los resultados producidos por T2 sean descartados de la BD, de manera que el saldo de la cuenta sea el que había antes de empezar la ejecución de T2. En consecuencia, T1 ha recuperado un valor que oficialmente nunca ha existido y ha visto interferida su ejecución por la transacción T2. Si las transacciones T1 y T2 hubieran estado aisladas correctamente, T1 nunca habría recuperado el valor provisional dejado por T2 y que finalmente ha sido descartado.

En el ejemplo previo, la interferencia de lectura no confirmada se produce a causa de la cancelación de la transacción que modifica los datos. Sin embargo, la interferencia se puede producir igualmente en el caso de que la transacción que modifica datos confirme los resultados.

Imaginemos ahora que tenemos dos transacciones, T1 y T2. La transacción T1 hace la consulta de un saldo de una cuenta corriente, mientras que T2 efectúa un par de reintegros de la misma cuenta corriente. Supongamos que el orden de ejecución de las operaciones es el que se muestra a continuación y que el SGBD no efectúa ningún control sobre el orden de ejecución de estas operaciones:

Transacción T1 (consulta saldo)	Transacción T2 (dos reintegros de 50)
	saldo := leer_saldo(cuenta)
	escribir_saldo(cuenta, saldo – 50)
saldo := leer_saldo(cuenta)	
COMMIT	
	saldo := leer_saldo(cuenta)
	escribir_saldo(cuenta, saldo – 50)
	COMMIT

En este caso, y aunque la transacción T2 confirma los resultados, T1 ve interferida su ejecución por T2 y recupera un dato provisional pendiente de confirmación. Este dato corresponde a un saldo provisional para la cuenta corriente que corresponde al saldo que queda después del primer reintegro.

3) Lectura no repetible. Esta interferencia se produce cuando una transacción, por los motivos que sea, necesita leer dos veces un mismo dato y en cada lectura recupera un valor diferente por el hecho de que hay otra transacción que se ejecuta simultáneamente y que efectúa una modificación del dato leído. Esto podría pasar en diversas situaciones, como la que se muestra a continuación:

Transacción T1 (reintegro de 20)	Transacción T2 (consulta saldo)
	saldo := leer_saldo(cuenta)
saldo := leer_saldo(cuenta)	
escribir_saldo(cuenta, saldo – 20)	
	saldo := leer_saldo(cuenta)
COMMIT	
	COMMIT

La transacción T2, que consulta dos veces el saldo de una misma cuenta corriente, recupera en cada lectura un valor diferente por el hecho de que la transacción T1, entre las dos lecturas, efectúa un reintegro e interfiere en la ejecución de la transacción T2. Si las transacciones se hubieran aislado correctamente entre ellas, T2 habría recuperado el mismo valor para el saldo de cuenta corriente en las dos lecturas: o bien habría recuperado el valor que correspondiera al saldo de la cuenta antes de que se efectuara el reintegro de T1, o bien, al saldo que quedara después de que se efectuara el reintegro de T1.

4) Análisis inconsistente (y el caso particular de fantasmas). Los tres tipos de interferencias anteriores se producen con respecto a un único dato de la BD, es decir, ocurren cuando dos transacciones intentan acceder a un mismo ítem de datos y, como mínimo, una de las dos transacciones modifica este ítem de datos. Pero también puede haber interferencias con respecto a la visión que dos transacciones tienen de un conjunto de datos que están interrelacionados.

Esto puede pasar, por ejemplo, cuando una transacción T1 lee unos datos mientras que otra transacción T2 actualiza una parte de ellos.

T1 puede obtener un estado de los datos incorrecto, como sucede con las siguientes transacciones:

Transacción T1 (consulta de saldos)	Transacción T2 (transferencia)
saldo2 := leer_saldo(cuenta2)	
	saldo1 := leer_saldo(cuenta1)
	escribir_saldo(cuenta1, saldo1 – 100)
saldo1 := leer_saldo(cuenta1)	
COMMIT	
	saldo2 := leer_saldo(cuenta2)
	escribir_saldo(cuenta2, saldo2 + 100)
	COMMIT

Los saldos que lee T1 no son correctos. No se corresponden ni con los de antes de la transferencia entre las dos cuentas ni con los de después, sino a un estado intermedio de T2 que no se tendría que haber visto nunca fuera del ámbito de T2. En consecuencia, T1 ha visto interferida su ejecución por T2.

Un caso particular bastante frecuente de esta interferencia son los **fantasmas**. Esta interferencia se puede producir cuando una transacción lee un conjunto de datos relacionado y hay otra transacción que dinámicamente cambia este conjunto de datos de interés añadiendo nuevos datos. Básicamente, la interferencia ocurre cuando se produce la siguiente secuencia de acontecimientos:

- Una transacción T1 lee una serie de datos que cumplen una condición C determinada.

- Una transacción T2 inserta nuevos datos que cumplen la condición C, o bien actualiza datos que no satisfacían la condición C y que ahora sí que la satisfacen.

- La transacción T1 vuelve a leer los datos que satisfacen la condición C o bien alguna información que depende de estos datos.

La consecuencia de esto es que T1 ve interferida su ejecución por T2 y encuentra un fantasma, es decir, unos datos que antes no cumplían la condición y que ahora sí que la cumplen. O también podría pasar que T1 no viera el fantasma directamente, sino el efecto que tiene en otros datos, tal como muestran los siguientes ejemplos:

Transacción T1 (lista de cuentas)	Transacción T2 (creación de cuentas)
leer todas las cuentas del banco. Imaginemos que sólo tenemos tres cuentas (cuenta1, cuenta2 y cuenta3) mostrar datos cuenta1 mostrar datos cuenta2 mostrar datos cuenta3	

Transacción T1 (lista de cuentas)	Transacción T2 (creación de cuentas)
	crear_cuenta(cuenta4)
	saldo_inicial(cuenta4, 100)
	COMMIT
sumar el saldo de todas las cuentas obtenemos saldo cuenta1 + saldo cuenta2 + saldo cuenta3 + 100 (la cuenta4 con saldo 100 es el fantasma)	
COMMIT	

En este primer ejemplo, la cuenta 4 es un fantasma desde el punto de vista de T1. Y además T1 ve el efecto que tiene en la suma de saldos que se produce y que, desde su punto de vista, da un resultado incoherente. Si T1 hubiera estado aislada correctamente de la transacción T2, una vez ejecutada la primera consulta, nunca tendría que haber encontrado los datos correspondientes a la cuenta 4.

Finalmente, el siguiente ejemplo muestra un fantasma que se produce a consecuencia de una actualización de datos por parte de la transacción T2. Imaginemos que los propietarios de las cuentas 1 y 2 viven en Barcelona; los propietarios de la cuenta 3 residen en Madrid, y los titulares de la cuenta 4, que vivían en Tarragona, notifican que ahora residirán en Barcelona.

Transacción T1 (lista de cuentas clientes de Barcelona)	Transacción T2 (cambio residencia)
leer cuentas clientes Barcelona mostrar datos cuenta1 mostrar datos cuenta2	
	cambiar_residencia(cuenta4, Barcelona)
sumar el saldo de todas las cuentas de clientes de Barcelona obtenemos saldo cuenta1 + saldo cuenta2 + saldo cuenta4 (la cuenta4 es el fantasma)	
COMMIT	
	COMMIT

En el ejemplo anterior, la cuenta 4 es nuevamente un fantasma desde el punto de vista de la transacción T1.

5.4. Nivel de concurrencia

Un SGBD puede resolver los problemas de interferencias entre transacciones que hemos visto anteriormente de dos maneras:

a) Cancelar automáticamente (en inglés, *abort*) las transacciones problemáticas y deshacer los cambios que han podido producir sobre la BD.

b) Suspender la ejecución de una de las transacciones problemáticas temporalmente y retomarla cuando haya desaparecido el peligro de interferencia. En algunos casos, esta situación también puede comportar la cancelación de transacciones.

Las dos soluciones implican un coste en términos de disminución del rendimiento de la BD. Precisamente, en lo relativo a la gestión de transacciones, uno de los objetivos de los SGBD es minimizar estos efectos negativos.

Se denomina **nivel de concurrencia** al grado de aprovechamiento de los recursos de proceso disponibles, según el solapamiento de la ejecución de las transacciones que acceden de forma concurrente a la BD y se confirman.

El objetivo del SGBD es aumentar el trabajo efectivo (es decir, el trabajo realmente útil para los usuarios) efectuado por unidad de tiempo. Sin duda, las transacciones que suspenden su ejecución no hacen trabajo efectivo, y todavía menos lo hacen las transacciones que finalmente cancelan su ejecución.

Uno de los grandes retos de la gestión de transacciones es alcanzar el nivel de concurrencia adecuado. Esto se consigue intentando que no se produzcan cancelaciones o suspensiones de ejecución de las transacciones cuando no es realmente necesario para impedir una interferencia. Desgraciadamente, este objetivo nunca se satisface del todo, ya que implicaría un esfuerzo excesivo y sería perjudicial para el rendimiento global por otros motivos. Los SGBD intentan obtener un compromiso óptimo entre el nivel de concurrencia que permiten y el coste que esto comporta en términos de tareas de control.

5.5. Visión externa de las transacciones

SQL estándar fuerza a que, una vez que se haya establecido una conexión con la BD, la primera sentencia SQL que queramos ejecutar mediante SQL interactivo **implícitamente** inicie la ejecución de una transacción. Una vez iniciada la transacción, esta permanecerá activa hasta que **explícitamente** y de una manera obligatoria indiquemos su finalización.

Última versión de SQL estándar

Cuando hablemos de las sentencias SQL, siempre nos referiremos a la última versión de SQL estándar, ya que tiene como subconjunto todas las anteriores y, por lo tanto, todo lo que era válido en la anterior lo continuará siendo en la siguiente. Solo especificaremos el año de una versión de SQL cuando queramos enfatizar que se hizo una aportación determinada concretamente en esa versión.

Por defecto, SQL estándar fuerza que esta transacción nunca vea interferida su ejecución y que tampoco pueda interferir en la ejecución de otras transacciones. En definitiva, por defecto, el SGBD deberá garantizar el correcto aislamiento de todas las transacciones que accedan de forma concurrente a la BD.

Para informar sobre las características asociadas a una transacción desde SQL:1992, disponemos de la siguiente sentencia:

```
SET TRANSACTION modo_acceso;
```

en la que `modo_acceso` puede ser `READ ONLY`, en el caso de que la transacción solo consulte la BD, o `READ WRITE`, en el caso de que la transacción modifique la BD.

La sentencia previa solo se puede ejecutar en el caso de que no haya ninguna transacción en ejecución en la sesión de trabajo establecida con la BD; si hay alguna, SQL estándar especifica que el SGBD debería reportar una situación de error. Adicionalmente, las características especificadas serán aplicables al resto de transacciones que se ejecuten posteriormente durante la sesión de trabajo.

Para indicar la finalización de una transacción, SQL estándar nos ofrece la sentencia siguiente:

```
{COMMIT | ROLLBACK} [WORK];
```

Mientras que `COMMIT` confirma todos los cambios producidos contra la BD durante la ejecución de la transacción, `ROLLBACK` los deshace y deja la BD como estaba antes de que se iniciara la transacción. La palabra reservada `WORK` solo sirve para explicar qué hace la sentencia y es opcional.

Ejemplos de uso de la sentencia `SET TRANSACTION`

Supongamos que tenemos una BD de un banco que guarda datos de las cuentas de los clientes. En concreto, consideremos que tenemos la siguiente tabla (clave primaria subrayada):

```
cuentas(num_cuenta, tipo_cuenta, saldo, comision)
```

Considerando que hemos establecido la conexión con la BD, podemos ejecutar las siguientes sentencias durante nuestra sesión de trabajo con los efectos que se comentan:

Sentencia	Comentarios
`SET TRANSACTION READ WRITE;`	Informamos de que las transacciones que se ejecutarán en la sesión establecida pueden hacer lecturas y cambios en la BD.

Sentencia	Comentarios

`UPDATE cuentas`
`SET saldo=saldo*1.10`
`WHERE num_cuenta="234509876";`
	Se inicia la ejecución de una transacción que puede hacer lecturas y cambios en la BD.
	Incrementamos en un 10% el saldo de la cuenta número 234509876.

`SELECT *`
`FROM cuentas`
`WHERE num_cuenta="234509876";`
	Recuperamos los datos de la cuenta número 234509876.

`COMMIT;`	Confirmamos los resultados producidos por la transacción.

`UPDATE cuentas`
`SET saldo=saldo-500`
`WHERE num_cuenta="234509876";`
	Esta sentencia inicia implícitamente la ejecución de una nueva transacción que puede hacer lecturas y cambios en la BD.
	Transferimos 500 € de la cuenta 234509876 a la cuenta 987656574. Suponemos que hay suficiente saldo.
	Disminuimos el saldo de la cuenta de origen.

`UPDATE cuentas`
`SET saldo=saldo+500`
`WHERE num_cuenta="987656574";`
	Incrementamos el saldo de la cuenta de destino.

`COMMIT;`	Confirmamos los resultados producidos por la transacción.

`SELECT saldo`
`FROM cuentas`
`WHERE tipo_cuenta="ahorro a plazo";`
	Esta sentencia inicia implícitamente la ejecución de una nueva transacción que puede hacer lecturas y cambios en la BD.
	Consultamos el saldo de las cuentas de un tipo determinado.

`SET TRANSACTION READ ONLY;`	Esta sentencia genera un error que nos será reportado por el SGBD. No podemos cambiar las características de las transacciones porque ya tenemos una transacción en ejecución.
`ROLLBACK;`	Como se ha producido un error, cancelamos la transacción.
`SET TRANSACTION READ ONLY;`	Informamos de que las transacciones que se ejecuten en la sesión de trabajo a partir de este momento sólo leerán la BD. Además, la sentencia también inicia la ejecución de una transacción.

`SELECT saldo`
`FROM cuentas`
`WHERE tipo_cuenta="ahorro a plazo";`
	Consultamos el saldo de las cuentas de un tipo determinado.

`COMMIT;`	Confirmamos los resultados producidos por la transacción.

El comienzo implícito de transacciones, en un entorno de aplicación real, puede crear confusiones sobre el alcance de cada transacción, si este alcance no se documenta correctamente. Por esto, desde SQL:1999 se propone utilizar la siguiente sentencia:

```
START TRANSACTION [modo_acceso];
```

en la que `modo_acceso` puede ser `READ ONLY` o `READ WRITE`. Si no se especifica el modo de acceso, la sentencia simplemente inicia la ejecución de una nueva transacción, de acuerdo con las características que se hayan especificado previamente. Si antes no se ha especificado ninguna característica, SQL estándar enuncia que la transacción se tiene que considerar de tipo `READ WRITE`.

Inicio de las transacciones

Muchos SGBD incorporan sentencias propias para marcar de forma explícita el inicio de las transacciones. En la mayoría de los casos, esta sentencia es `BEGIN WORK` o, simplemente, `BEGIN`, porque la palabra clave `WORK` es opcional.

Ejemplos de uso de la sentencia `START TRANSACTION`

En la BD del ejemplo anterior, y asumiendo que hemos establecido la conexión con la BD, podemos ejecutar las siguientes sentencias con los efectos que se comentan:

Sentencia	Comentarios
`START TRANSACTION READ ONLY;`	Informamos de que empieza la ejecución de una transacción de sólo lectura.

`SELECT saldo`
`FROM cuentas`
`WHERE tipo_cuenta="ahorro a plazo";`
	Consultamos el saldo de las cuentas de un tipo determinado.

`COMMIT;`	Confirmamos los resultados producidos por la transacción.
`START TRANSACTION READ WRITE;`	Se inicia la ejecución de una transacción que puede consultar y modificar la BD.

`UPDATE cuentas`
`SET saldo=saldo*1.10`
`WHERE num_cuenta="234509876";`
	Incrementamos en un 10% el saldo de la cuenta número 234509876.

`SELECT *`
`FROM cuentas`
`WHERE num_cuenta="234509876";`
	Recuperamos los datos de la cuenta número 234509876.

`COMMIT;`	Confirmamos los resultados producidos por la transacción.
`START TRANSACTION;`	Inicio de una nueva transacción. No se indican sus características. Por lo tanto, se aplican las especificadas anteriormente. En consecuencia, la transacción puede consultar la BD y modificarla.

`UPDATE cuentas`
`SET saldo=saldo-500`
`WHERE num_cuenta="234509876";`
	Transferencia bancaria.
	Disminuimos el saldo de la cuenta de origen.

`UPDATE cuentas`
`SET saldo=saldo+500`
`WHERE num_cuenta="987656574";`
	Incrementamos el saldo de la cuenta de destino.

`COMMIT;`	Confirmamos los resultados producidos por la transacción.

`SELECT saldo`
`FROM cuentas`
`WHERE tipo_cuenta="ahorro a plazo";`
	Esta sentencia inicia implícitamente la ejecución de una nueva transacción que puede hacer lecturas y cambios en la BD. Por lo tanto, no es obligatorio marcar explícitamente el inicio de las transacciones, aunque sea conveniente. De esta manera se asegura la compatibilidad con las versiones previas del estándar.
	Consultamos el saldo de las cuentas de un tipo determinado.

`COMMIT;`	Confirmamos los resultados producidos por la transacción.

5.5.1. Relajación del nivel de aislamiento

Hasta ahora habíamos considerado que siempre era necesario garantizar una protección total ante cualquier tipo de interferencias. No obstante, esta protección total exige una sobrecarga del SGBD en términos de gestión de información de control y una disminución del nivel de concurrencia.

En determinadas circunstancias, es conveniente relajar el nivel de aislamiento y posibilitar que se produzcan interferencias. Esto es correcto si se sabe que estas interferencias no ocurrirán realmente o si en el entorno de aplicación en el que nos encontramos no es importante que se produzcan.

Si nos centramos en SQL estándar, las instrucciones `SET TRANSACTION` y `START TRANSACTION` permiten relajar el nivel de aislamiento. Tienen la siguiente sintaxis:

```
SET TRANSACTION {READ ONLY | READ WRITE},
ISOLATION LEVEL nivel_aislamiento;

START TRANSACTION [{READ ONLY | READ WRITE}],
ISOLATION LEVEL nivel_aislamiento;
```

en la que nivel_aislamiento puede ser `READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ` o `SERIALIZABLE`.

El **nivel de aislamiento** determina las interferencias que pueden desencadenar otras transacciones en la transacción que empieza. De acuerdo con los tipos de interferencia que hemos descrito, la siguiente tabla indica las que se evitan con cada nivel de aislamiento:

	Actualización perdida	**Lectura no confirmada**	**Lectura no repetible y análisis inconsistente (excepto fantasmas)**	**Fantasmas**
`READ UNCOMMITTED`	Sí	No	No	No
`READ COMMITTED`	Sí	Sí	No	No
`REPEATABLE READ`	Sí	Sí	Sí	No
`SERIALIZABLE`	Sí	Sí	Sí	Sí

En ella los niveles aparecen de menos a más estrictos y, por lo tanto, de menos a más eficientes:

1) El nivel `READ UNCOMMITTED` protege los datos actualizados y evita que ninguna otra transacción los actualice hasta que no se acabe la transacción. No ofrece ninguna garantía con respecto a los datos que lea la transacción. Pueden ser datos actualizados por una transacción que todavía no ha confirmado y, además, otra transacción los puede actualizar inmediatamente.

2) El nivel `READ COMMITTED` protege parcialmente las lecturas e impide que la transacción lea los datos actualizados por otra transacción que todavía no se hayan confirmado.

3) El nivel `REPEATABLE READ` impide que otra transacción actualice un dato que haya leído la transacción hasta que esta no se acabe. De esta manera, la transacción puede volver a leer este dato sin riesgo de que lo hayan cambiado.

4) El nivel `SERIALIZABLE` ofrece un aislamiento total y evita cualquier tipo de interferencias, incluyendo los fantasmas. Esto significa que no solamente protege los datos que haya visto la transacción, sino también cualquier información de control que se haya utilizado para hacer búsquedas.

La definición del SQL estándar establece que un SGBD concreto tiene la obligación de garantizar como mínimo el nivel de aislamiento que la transacción haya solicitado, aunque puede optar por ofrecer un aislamiento más restrictivo. Por lo tanto, el único nivel que un SGBD tiene la obligación de implementar es el más alto, el `SERIALIZABLE`.

5.5.2. Responsabilidades del SGBD y del desarrollador

Hemos visto qué es una transacción y qué propiedades tiene que cumplir. Examinemos la contribución del SGBD para conseguir garantizar estas propiedades y los aspectos que dependen del desarrollador de las aplicaciones.

Horario

La ejecución concurrente de un conjunto de transacciones (en las que se preserva el orden de las operaciones dentro de cada transacción) recibe el nombre de horario o historia.

1) Responsabilidades del SGBD

a) Conseguir que el horario que se produzca a medida que el SGBD recibe peticiones de lectura o escritura, y de `COMMIT` o `ROLLBACK` de las transacciones que se ejecuten de forma concurrente sobre la BD, sea correcto (sin interferencias). Naturalmente, en el caso de que se haya relajado el nivel de aislamiento para algunas transacciones, será necesario que el SGBD considere correctos más horarios.

El SGBD consigue horarios libres de interferencias, sobre todo, de dos maneras (no necesariamente excluyentes entre ellas): cancelando automáticamente las transacciones problemáticas o suspendiendo la ejecución de la transacción hasta que la pueda retomar sin problemas. El conjunto de mecanismos que se responsabiliza de estas tareas se llama **control de concurrencia**.

Es necesario que estos mecanismos sean tan transparentes a la programación como sea posible, de manera que no se añadan dificultades innecesarias al desarrollo. No obstante, a veces es necesario ofrecer servicios[1] que modifiquen el comportamiento por defecto del SGBD para aumentar el nivel de concurrencia.

(1)Por ejemplo, la posibilidad de relajar el nivel de aislamiento de las transacciones.

b) Comprobar que los cambios que ha hecho una transacción verifican todas las reglas de integridad que se han definido en la BD. Esto se puede hacer justo antes de aceptar el `COMMIT` de la transacción, rechazándolo si se viola alguna regla, o inmediatamente después de que se ejecute cada petición dentro de la transacción.

c) Impedir que en la BD permanezcan cambios de transacciones que no se llegan a confirmar y que se pierdan los cambios que han llevado a cabo transacciones confirmadas en el caso de que se produzcan cancelaciones de transacciones, caídas del SGBD o de las aplicaciones, desastres (como incendios) o fallos de los dispositivos externos de almacenaje. En general, hablamos de **recuperación** para referirnos al conjunto de mecanismos que se encargan de estas tareas.

2) Tareas del desarrollador de aplicaciones

a) Identificar con precisión las transacciones de una aplicación, es decir, el conjunto de operaciones que necesariamente se tiene que ejecutar de una manera atómica sobre la BD de acuerdo con los requerimientos de los usuarios.

En este sentido, las transacciones tendrían que durar el mínimo imprescindible. En concreto, puede ser muy peligroso que una aplicación tenga una transacción en ejecución mientras se espera la entrada de información por parte del usuario. A veces, los usuarios pueden tardar bastante rato en proporcionar ciertos datos o, simplemente, en apretar el botón de aceptación de un mensaje. Incluso es posible que cualquier circunstancia les haga dejar a medias lo que hacían y que la aplicación se quede bastante tiempo a la espera de que el usuario vuelva a ella. Hasta que el usuario no permite que la transacción se acabe, esta puede impedir la actualización o incluso la lectura de los datos a los que ya haya accedido. Esto significa más gasto de recursos y un freno importante en el nivel de concurrencia posible. Por lo tanto, y siempre que sea posible, se suele recomendar que durante una transacción no se pare nunca la ejecución de la aplicación a la espera de que se produzca una actuación determinada por parte del usuario.

b) Garantizar que las transacciones mantienen la consistencia de la BD de acuerdo con los requerimientos de los usuarios y teniendo en cuenta las restricciones de integridad y los disparadores definidos en la BD.

c) Considerar aspectos de rendimiento. En particular, el desarrollador tiene que ser capaz de estudiar y mejorar el nivel de concurrencia de acuerdo con los conocimientos que tenga de los mecanismos de control de concurrencia del SGBD y las posibilidades de modificar su funcionamiento.

5.6. Transacciones en PostgreSQL

Por defecto, y si no se indica expresamente lo contrario, PostgreSQL trabaja con transacciones implícitas (este modo de trabajo también se conoce con el nombre de *autocommit* activado). Esto quiere decir que cualquier grupo de sentencias SQL que seleccionemos (por ejemplo, desde el PgAdmin) y enviemos a ejecutar será tratado como una transacción. Si el grupo de sentencias enviado no genera ningún error, los resultados pasarán a ser definitivos en la BD. De lo contrario, los resultados serán descartados por el SGBD.

Ya sabemos que trabajar con transacciones implícitas en un entorno de aplicación real puede crear confusiones sobre el alcance de cada transacción. Por esto, PostgreSQL nos ofrece la sentencia de SQL estándar `START TRANSACTION`, y también una sentencia propia, la sentencia `BEGIN`, para indicar de forma explícita el comienzo de una transacción. Cuando se indica explícitamente el comienzo de una transacción, PostgreSQL desactiva la modalidad *autocommit* y la transacción permanecerá activa hasta que confirmemos o cancelemos sus resultados de forma explícita. Para indicar la finalización de la transacción, disponemos de las sentencias de SQL estándar `COMMIT` y `ROLLBACK`.

Adicionalmente, también tenemos disponible la sentencia `SET TRANSACTION` de SQL estándar para indicar las características de la transacción (si es `READ ONLY` o `READ WRITE` y el nivel de aislamiento) en el caso de que no se haya hecho anteriormente; por ejemplo, con las sentencias `START TRANSACTION` o `BEGIN`. Si el usuario no ha especificado ninguna característica para las transacciones que quiere ejecutar, por defecto, PostgreSQL considerará que son transacciones `READ WRITE` que trabajan con un nivel de aislamiento `READ COMMITTED`.

Aunque PostgreSQL permite especificar cualquiera de los niveles de aislamiento propuestos por SQL estándar (`READ UNCOMMITTED`, `READ COMMITTED`, `REPEATABLE READ` y `SERIALIZABLE`), de hecho, internamente únicamente trabaja con dos niveles de aislamiento. Estos niveles son los de `READ COMMITTED` y `SERIALIZABLE`:

1) El nivel de aislamiento `READ COMMITTED` evita que la transacción se vea involucrada en interferencias de actualización perdida y de lectura no confirmada. La transacción se podría ver implicada en interferencias de lectura no repetible y de análisis inconsistente (incluyendo fantasmas).

2) Por su parte, el nivel de aislamiento `SERIALIZABLE` evita cualquier tipo de interferencia.

Nota

Si indicamos un nivel de aislamiento `READ UNCOMMITTED`, PostgreSQL, internamente, lo transformará en `READ COMMITTED`.

Si indicamos un nivel de aislamiento `REPEATABLE READ`, PostgreSQL, internamente, lo transformará en `SERIALIZABLE`.

El hecho de que PostgreSQL solo tenga que considerar internamente dos niveles de aislamiento está relacionado con el mecanismo para el control de concurrencia que implementa. Este mecanismo se basa en lo que se conoce como modelo de control de concurrencia multiversión (en inglés, *multiversion concurrency control*, abreviado MVCC).

5.7. Importancia de las transacciones en OLTP frente a OLAP

Para finalizar, es importante destacar la importancia de las transacciones en sistemas OLTP en comparación con los sistemas OLAP. Para ello, utilizaremos como base la siguiente tabla con las principales diferencias entre estos dos sistemas.

	OLTP	OLAP
Uso	Específico de la aplicación	Soporte a la decisión
Carga de trabajo	Predefinida	Impredecible
Acceso	Lectura/Escritura	Lectura
Complejidad de la consulta	Simple	Compleja
Filas por operación	Decenas/Cientos	Miles/Millones
Número de usuarios	Miles/Millones	Decenas/Cientos

Los sistemas OLTP están concebidos para resolver problemas concretos (tienen un propósito específico) y para ser utilizado en el día a día de las empresas, por lo que la carga de trabajo suele estar claramente predefinida. En estos sistemas, el acceso a los datos se realiza tanto para lectura como para escritura (inserción de datos nuevos y actualizaciones de datos existentes), con una complejidad de consultas por lo general simple (pocas tablas, combinaciones y agrupaciones) que manejan un conjunto de datos relativamente pequeño. Por ejemplo, actualizar el nombre de un producto, actualizar el *stock* de un producto en concreto, obtener el listado de productos de una categoría en concreto, etc. Por último, el número de usuarios suele ser del orden de los miles o millones.

MVCC

El modelo de control de concurrencia MVCC no solo se halla disponible en PostgreSQL, sino que también lo implementan otros SGBD, como por ejemplo, Oracle y SQL Server.

Para más información acerca del modelo MVCC de PostgreSQL, visitad el siguiente enlace:

http://www.postgresql.org/docs/9.3/static/mvcc.html

OLTP

On-line transaction processing

OLAP

On-line analytical processing

Por lo tanto, como ya hemos visto durante este capítulo y dadas las características de los entornos OLTP mencionadas, el uso de las transacciones resulta crítico para garantizar la consistencia de los datos y evitar así anomalías derivadas del acceso simultáneo de usuarios a la misma BD, o bien derivadas de fallos o desastres.

En contraposición, los sistemas OLAP están pensados para proporcionar análisis y dar soporte a la decisión, lo que significa que tienen una carga de trabajo impredecible, dependiendo de las necesidades del usuario final: por ejemplo, a finales de año la carga de trabajo de estos sistemas podría incrementarse debido a la cantidad de información a procesar y la necesidad de, por ejemplo, realizar informes *YoY*. El acceso a la información en estos sistemas es de lectura exclusivamente, por lo que un usuario final de OLAP nunca realizará actualizaciones en los datos. Además, como podremos suponer, la complejidad de las consultas a la BD es mucho mayor respecto de los sistemas OLTP, ya que requiere generalmente de muchas agrupaciones y operaciones de combinación con muchas tablas, manejando grandes volúmenes de datos, como ya hemos comentado, históricos. Por último, el número de usuarios, al contrario que en OLTP, suele ser de unas decenas/cientos. Esto es así porque los usuarios finales son aquellos que requieren de información para realizar análisis y tomar decisiones, siendo estos generalmente analistas de negocio y directivos de empresa.

YoY

Year over year, comparación de métricas correspondientes al año actual respecto al año anterior.

En consecuencia, al contrario que en OLTP, en los sistemas OLAP las transacciones no son necesarias para garantizar que los usuarios obtengan los datos consultados y, por ello, no tiene sentido hablar de transacciones en estos entornos, siempre desde un punto de vista del usuario final.

En cambio, es importante destacar que las transacciones sí son útiles (y necesarias en muchos casos) a la hora de desarrollar los procesos ETL que proporcionan información a los sistemas OLAP. Estos procesos ETL se encargan de leer información de los sistemas origen (que, entre otros, suelen ser los propios sistemas OLTP), información que, a través de tareas de transformación y de limpieza de datos, se carga en los sistemas OLAP. En estos casos, una transacción nos puede resultar de utilidad para, por ejemplo, garantizar la existencia de datos (recientes o no) tras un proceso de carga de datos. Por ejemplo, si una tabla se carga mediante un procedimiento de borrado y recarga (es decir, se eliminan todos los datos de la tabla y se vuelve a cargar de nuevo con el contenido más reciente), podríamos necesitar una transacción para garantizar que:

1) Los datos antiguos han sido eliminados y los nuevos han sido cargados, en el caso de que la carga de estos últimos haya terminado correctamente sin errores (datos cargados y confirmados mediante una operación de `COMMIT`).

2) En el caso de que haya sucedido un error en la carga de datos, los datos antiguos que se han borrado vuelvan a estar disponibles para evitar que los usuarios se queden sin información (cancelación de los cambios mediante una operación de `ROLLBACK`).

Resumen

Los contenidos de este módulo han sido creados con el objetivo de proporcionar al estudiante de un conjunto de conocimientos avanzados de SQL para el tratamiento de datos, orientándolos a entornos *data warehouse*.

El módulo comienza presentando el concepto de clave subrogada, indicando su objetivo y los beneficios que este tipo de mecanismo aporta. Se continúa con las *common table expression*, describiendo su funcionalidad y la estructura de este tipo de consultas, y la posibilidad de implementar recursividad mediante este método a través de un ejemplo práctico. Continuamos con el concepto de funciones analíticas y su utilización en consultas SQL, realizando también una introducción de aquellas funciones más importantes y utilizadas. Revisamos también la problemática de los valores nulos en BD operacionales y almacenes de datos, y las soluciones a cada uno de los problemas presentados. Para finalizar, introducimos el concepto de transacción y sus propiedades, además de cómo los SGBD implementan la gestión de transacciones para garantizar la integridad de la BD ante accesos simultáneos de múltiples usuarios.

Todos los conceptos presentados se han descrito de forma teórica, utilizando ejemplos prácticos para su explicación, y posteriormente se ha indicado cómo aplicarlos utilizando el SGBD PostgreSQL. También se añade la aplicación de estos conceptos utilizando el SGBD Oracle como anexo al módulo.

En el caso de que algún estudiante quiera profundizar en alguno de los conceptos estudiados en este módulo, puede consultar las referencias que hemos ido indicando a lo largo del módulo o los libros indicados en el apartado de bibliografía.

Ejercicios de autoevaluación

1. Suponed que tenemos una dimensión Hora, en la que cada fila representa una hora en concreto con una precisión de horas, minutos y segundos. ¿Consideráis válida la opción de crear una clave subrogada con significado en la que el valor de dicha clave subrogada sea un entero con formato HHMMSS (donde HH es hora, MM minutos y SS segundos)? ¿Podéis pensar en alguna otra manera de generar una clave subrogada con significado para esta dimensión?

2. Dada la siguiente tabla que gestiona el histórico de puestos de empleados (Histórico) con los siguientes datos a día de hoy:

Histórico				
id_empleado	**fecha_alta**	**nombre**	**puesto**	**salario_anual**
1	01-01-2015	Manuel Vázquez	Ingeniero de Software	23500
2	02-01-2015	Elena Rodríguez	Analista de Sistemas	16000
3	03-01-2015	José Pérez	Programador SQL	17000
1	01-03-2015	Manuel Vázquez	Jefe de Proyectos	25500
1	01-10-2015	Manuel Vázquez	Jefe de Proyectos Sénior	26500
3	15-06-2015	José Pérez	Programador SQL Sénior	19000
6	01-04-2015	Fernando Nadal	Becario	13000
2	01-10-2015	Elena Rodríguez	Responsable Analistas	24500
8	01-11-2015	Víctor Anllada	Jefe Departamento	28000

Generad, mediante el uso de CTE, la consulta SQL necesaria para obtener el nombre de empleados existentes a día 1 Septiembre 2015, su puesto y salario en dicho día, el puesto y salario actual, y la diferencia entre el salario actual y el salario a día 1 Septiembre 2015.

3. Utilizando la tabla de Histórico de puestos de empleados del ejercicio 2, obtened la misma información propuesta en dicho ejercicio, añadiendo al resultado el *ranking* de estos empleados sobre la base del salario anual en la fecha especificada, el *ranking* sobre la base del salario anual a día de hoy, y para cada empleado, la media de salario en Septiembre y actual para el conjunto de empleados activos en Septiembre 2015.

4. Supongamos que hemos diseñado una dimensión Producto con las siguientes columnas: id producto (clave subrogada, tipo entero), código producto (tipo alfanumérico), nombre producto (tipo alfanumérico), categoría producto (tipo alfanumérico), fecha de alta (tipo fecha), fecha de baja (tipo fecha). Suponed que las fechas de alta y baja se utilizan para gestionar el histórico de productos. Proporcionad el resultado final de realizar las siguientes operaciones, asumiendo que:

- La dimensión está vacía en el momento de comenzar el ejercicio.
- La secuencia utilizada para generar la clave subrogada comienza con un valor 1 y se incrementa en 1.
- Todas las filas de la dimensión tienen que tener valores para todas las columnas (no se pueden insertar valores nulos).

1) Se inserta el siguiente producto:

Código Producto	Nombre Producto	Categoría Producto	Fecha de Alta	Fecha de Baja
ABCD	Galletas	NULL	01-01-2015	NULL

2) Se inserta el siguiente producto:

Código Producto	Nombre Producto	Categoría Producto	Fecha de Alta	Fecha de Baja
EFGH	Atún	Conservas	02-01-2015	NULL

3) Se actualiza la fecha de baja del siguiente producto:

Código Producto	Fecha de Baja
ABCD	10-01-2015

4) Se inserta el siguiente producto:

Código Producto	Nombre Producto	Categoría Producto	Fecha de Alta	Fecha de Baja
ABCD	Jamón Serrano	Embutido	15-01-2015	NULL

5. Un centro médico privado dispone de una BD para guardar toda la información sobre sus pacientes, los médicos que trabajan en el centro, las visitas médicas de los pacientes y las recetas que se prescriben en las visitas. Entre otras, la BD incluye las dos tablas que se describen a continuación. La clave primaria de cada tabla está subrayada y a no ser que se diga lo contrario, todas las columnas son obligatorias (no admiten valores nulos).

```
MEDICO (num_med, nombre_med, especialidad, dirección, ciudad, teléfono, sueldo, DNI)
```

Información sobre los médicos. De estos se guarda el número de colegiado del médico (num_med, que es clave primaria), el nombre (nombre_med), su especialidad (especialidad), la dirección (dirección) y ciudad donde vive (ciudad), teléfono (teléfono), sueldo (sueldo) y el número del documento nacional de identidad (DNI). El sueldo tiene que ser siempre superior a cero. El DNI es único para cada médico pero puede ser nulo si el médico no tiene.

Supongamos que esta tabla contiene las siguientes filas:

MEDICO							
num_med	nombre_med	especialidad	dirección	ciudad	teléfono	sueldo	DNI
236	Pere Ba	Medicina interna	Pi 23	Lleida	678999000	3000	23456777
412	Mar Ros	Medicina interna	Born 1	Barcelona	934566549	3500	56789444

```
PACIENTE (num_pac, nombre_pac, dirección, ciudad, DNI)
```

Información sobre los pacientes. De estos se guarda el número de paciente (num_pac, que es clave primaria), el nombre (nombre_pac), la dirección (dirección), ciudad (ciudad) y el número del documento nacional de identidad (DNI). El DNI es único para cada paciente pero puede ser nulo si el paciente no tiene.

Supongamos que esta tabla contiene las siguientes filas:

PACIENTE				
num_pac	**nombre_Pac**	**dirección**	**ciudad**	**DNI**
989	Mark Smith	Or 23	Barcelona	NULL

Imaginemos que sobre estas tablas se desea ejecutar dos transacciones (T1 y T2). Dichas transacciones desean ejecutar las operaciones que se indican seguidamente:

T1	T2
SELECT * FROM medico	UPDATE medico SET especialidad = 'Pediatría' WHERE num_med = 236
INSERT INTO paciente VALUES (123, 'Vicente Hijos', 'Alcalá 10', 'Madrid', '46789999')	COMMIT
SELECT * FROM medico	
COMMIT	

Suponiendo que el SGBD no aplica **ningún mecanismo de control de concurrencia**, responded a las siguientes preguntas:

a) Proponed, si es posible, un horario que incorpore todas las sentencias SQL de T1 y T2, y que no contenga ninguna interferencia.

b) Proponed, si es posible, un horario que incorpore todas las sentencias SQL de T1 y T2 y que contenga una (y solo una) interferencia. Tenéis que indicar el nombre de la interferencia que se está produciendo y el nivel mínimo de aislamiento necesario de SQL para evitarla.

c) Suponed ahora que tenemos una transacción T3 que se ejecuta de forma concurrente con la transacción T1. El contenido de las tablas es el que se muestra en el enunciado (como si las transacciones T1 y T2 no se hubieran ejecutado). Proponed, si es posible, un ejemplo de sentencia SQL a ejecutar por T3 que cause una interferencia de tipo fantasma.

T1	T3
SELECT * FROM medico	
	SENTENCIA SQL
INSERT INTO paciente VALUES (123, 'Vicente Hijos', 'Alcalá 10', 'Madrid', '46789999')	
	COMMIT
SELECT * FROM medico	
COMMIT	

Solucionario

1. La solución propuesta de utilizar un formato HHMMSS como clave subrogada de una dimensión Hora podría ser válida siempre y cuando las horas HH se representen en formato 24H, es decir, que se diferencien las horas entre las que son de madrugada/mañana (desde las 00:00:00 hasta las 11:59:59) y las que son de tarde (desde las 12:00:00 hasta las 23:59:59). Considerando este caso en concreto, las claves subrogadas se podrían construir de la siguiente forma:

Clave Subrogada	Hora Formato 24H	Hora Formato 12H	Hora Formato AM/PM
000000	00:00:00	12:00:00	12:00:00 AM
000001	00:00:01	12:00:01	12:00:01 AM
...	...	...	...
120000	12:00:00	12:00:00	12:00:00 PM
120001	12:00:01	12:00:01	12:00:01 PM
120002	12:00:02	12:00:02	12:00:02 PM
...	...	...	...
235958	23:59:58	11:59:58	11:59:58 PM
235959	23:59:59	11:59:59	11:59:59 PM

Cabe destacar que en este ejemplo, debido a que la clave subrogada es numérica entera, aquellas horas de madrugada (entre las 00:00:00 y las 09:59:59), que son las que comienzan con ceros, se representarán sin los ceros iniciales ya que el SGBD los elimina. Por lo tanto, estos casos se almacenarán en la BD de la siguiente manera. Este resultado sería el mismo que el de utilizar una secuencia numérica autogenerada que comience en cero, y asumiendo que las horas se han insertado en orden en la dimensión.

Clave Subrogada	Hora Formato 24H	Hora Formato 12H	Hora Formato AM/PM
0	00:00:00	12:00:00	12:00:00 AM
1	00:00:01	12:00:01	12:00:01 AM
2	00:00:02	12:00:02	12:00:02 AM
...	...	...	...
95959	09:59:59	09:59:59	09:59:59 AM
100000	10:00:00	10:00:00	10:00:00 AM
...	...	...	...

Otra forma de generar esta clave subrogada y seguir manteniendo un significado sin eliminar esos ceros iniciales sería añadir 1000000 al valor de la clave subrogada anterior, por lo que todas las horas tendrán un formato 1HHMMSS (1000000 serán las 00:00:00, 1150003 serán las 15:00:03, etc.).

Clave Subrogada	Hora Formato 24H	Hora Formato 12H	Hora Formato AM/PM
1000000	00:00:00	12:00:00	12:00:00 AM
1000001	00:00:01	12:00:01	12:00:01 AM
1000002	00:00:02	12:00:02	12:00:02 AM
...	...	...	...
1095959	09:59:59	09:59:59	09:59:59 AM
1100000	10:00:00	10:00:00	10:00:00 AM
...	...	...	...

2. La consulta propuesta para obtener los detalles especificados se puede ver a continuación:

```
WITH emp_sep2015 AS (
    SELECT
      id_empleado,
      MAX ( fecha_alta ) AS MaxFechaAlta
    FROM historico
    WHERE fecha_alta <= '2015-09-01'
    GROUP BY  id_empleado
  ), emp_actual AS (
    SELECT
      id_empleado,
      MAX ( fecha_alta ) AS MaxFechaAlta
    FROM historico
    WHERE id_empleado IN (
        SELECT
          eSep2015.id_empleado
        FROM
          emp_sep2015 eSep2015
      )
    GROUP BY id_empleado
  ), detalles_sep2015 AS (
    SELECT
      historico.id_empleado,
      historico.nombre,
      historico.puesto,
      historico.salario_anual
    FROM historico
    INNER JOIN emp_sep2015 eSep2015 ON
      eSep2015.id_empleado    = historico.id_empleado
    AND eSep2015.maxfechaalta = historico.fecha_alta
  ), detalles_actual AS (
    SELECT
      historico.id_empleado,
      historico.nombre,
      historico.puesto,
      historico.salario_anual
    FROM historico
    INNER JOIN emp_actual eActual ON
      eActual.id_empleado    = historico.id_empleado
    AND eActual.maxfechaalta = historico.fecha_alta
  )
SELECT
  dSep2015.nombre,
  dSep2015.puesto PuestoSep2015,
  dSep2015.salario_anual SalarioSep2015,
  dActual.puesto PuestoActual,
  dActual.salario_anual SalarioActual,
  dActual.salario_anual - dSep2015.salario_anual Diferencia
FROM
  detalles_sep2015 dSep2015
INNER JOIN detalles_actual dActual ON
 dSep2015.id_empleado = dActual.id_empleado;
```

3. La consulta propuesta para obtener los detalles especificados se puede ver a continuación:

```
WITH emp_sep2015 AS (
    SELECT
      id_empleado,
      MAX ( fecha_alta ) AS MaxFechaAlta
    FROM historico
    WHERE fecha_alta <= '2015-09-01'
    GROUP BY  id_empleado
  ), emp_actual AS (
    SELECT
      id_empleado,
      MAX ( fecha_alta ) AS MaxFechaAlta
    FROM historico
    WHERE id_empleado IN (
        SELECT
          eSep2015.id_empleado
        FROM
          emp_sep2015 eSep2015
      )
    GROUP BY id_empleado
  ), detalles_sep2015 AS (
    SELECT
      historico.id_empleado,
      historico.nombre,
      historico.puesto,
      historico.salario_anual
    FROM historico
    INNER JOIN emp_sep2015 eSep2015 ON
      eSep2015.id_empleado    = historico.id_empleado
    AND eSep2015.maxfechaalta = historico.fecha_alta
  ), detalles_actual AS (
    SELECT
      historico.id_empleado,
      historico.nombre,
      historico.puesto,
      historico.salario_anual
    FROM historico
    INNER JOIN emp_actual eActual ON
      eActual.id_empleado    = historico.id_empleado
    AND eActual.maxfechaalta = historico.fecha_alta
  )
SELECT
  dSep2015.nombre,
  dSep2015.puesto PuestoSep2015,
  dSep2015.salario_anual SalarioSep2015,
  dActual.puesto PuestoActual,
  dActual.salario_anual SalarioActual,
  dActual.salario_anual - dSep2015.salario_anual Diferencia,
  RANK() OVER (ORDER BY dSep2015.salario_anual DESC) AS rk_salarioSep2015,
  RANK() OVER (ORDER BY dActual.salario_anual DESC)  AS rk_salarioActual2015,
  AVG(dSep2015.salario_anual) OVER () mediaSalSep2015,
  AVG(dActual.salario_anual) OVER () mediaSalSActual
FROM detalles_sep2015 dSep2015
INNER JOIN detalles_actual dActual ON
  dSep2015.id_empleado = dActual.id_empleado;
```

4. El resultado de realizar las operaciones indicadas es el siguiente:

1) Añadimos el producto con clave subrogada 1, categoría DESCONOCIDO y fecha de baja 31-12-9999 (indicando que no se ha dado de baja y que el registro es válido hasta el fin de los tiempos).

Clave Subrogada	Código Producto	Nombre Producto	Categoría Producto	Fecha de Alta	Fecha de Baja
1	ABCD	Galletas	DESCONOCIDO	01-01-2015	31-12-9999

2) Añadimos el producto con clave subrogada 2 y fecha de baja 31-12-9999 (indicando que no se ha dado de baja y que el registro es válido hasta el fin de los tiempos).

Clave Subrogada	Código Producto	Nombre Producto	Categoría Producto	Fecha de Alta	Fecha de Baja
1	ABCD	Galletas	DESCONOCIDO	01-01-2015	31-12-9999
2	EFGH	Atún	Conservas	02-01-2015	31-12-9999

3) Se actualiza el producto existente con una nueva fecha de baja. Con esto conseguimos crear un período de tiempo en el que la fila es válida (entre el 01-01-2015 y 02-01-2015).

Clave Subrogada	Código Producto	Nombre Producto	Categoría Producto	Fecha de Alta	Fecha de Baja
1	ABCD	Galletas	DESCONOCIDO	01-01-2015	10-01-2015
2	EFGH	Atún	Conservas	02-01-2015	31-12-9999

4) Añadimos el producto con clave subrogada 3 y fecha de baja 31-12-9999 (indicando que no se ha dado de baja y que el registro es válido hasta el fin de los tiempos). Vemos que el código de producto es el mismo que el anterior, por lo que habría que insertarlo con una nueva clave subrogada para mantener el histórico de valores.

Clave Subrogada	Código Producto	Nombre Producto	Categoría Producto	Fecha de Alta	Fecha de Baja
1	ABCD	Galletas	DESCONOCIDO	01-01-2015	10-01-2015
2	EFGH	Atún	Conservas	02-01-2015	31-12-9999
3	ABCD	Jamón Serrano	Embutido	15-01-2015	31-12-9999

5.

a) Existen varias soluciones válidas. Una de las condiciones que se tienen que dar para que un horario pueda contener interferencias es que las transacciones se ejecuten de forma concurrente, es decir, solapada. Por lo tanto, por ejemplo, un horario donde las transacciones no solapen la ejecución de las sentencias que incorporan nunca presentará interferencias. Teniendo en cuenta esto, una posible solución sería la siguiente:

T1	T2
`SELECT * FROM medico`	
`INSERT INTO paciente VALUES (123, 'Vicente Hijos', 'Alcalá 10', 'Madrid', '46789999')`	
`SELECT * FROM medico`	
`COMMIT`	
	`UPDATE medico SET especialidad = 'Pediatría' WHERE num_med = 236`
	`COMMIT`

b) Un posible ejemplo de horario con una interferencia para T1 y T2 sería el que se muestra seguidamente. La interferencia que se produce es de lectura no repetible. La segunda `SELECT` ejecutada por T1 recupera las mismas filas que la primera `SELECT`, pero con valores diferentes en cada lectura (en concreto, la fila 1 de la tabla `MEDICO` que corresponde con el médico con número de médico 236 es la que ha cambiado su valor, en concreto la especialidad). El nivel mínimo de SQL que evitaría la interferencia es `REPEATABLE READ`.

T1	T2
`SELECT * FROM medico`	
	`UPDATE medico SET especialidad = 'Pediatría' WHERE num_med = 236`
	`COMMIT`
`INSERT INTO paciente VALUES (123, 'Vicente Hijos', 'Alcalá 10', 'Madrid', '46789999')`	

T1	T2
SELECT * FROM medico	
COMMIT	

c) Para que se produzca una interferencia de tipo fantasma entre T1 y T3 es necesario que T3 extienda el conjunto de datos a recuperar por las sentencias SELECT que ejecuta la transacción T1. Esto podría pasar con sentencias de INSERT y UPDATE. Dado que las sentencias SELECT de T1 no incorporan cláusula WHERE, la única posibilidad sería que T3 ejecute una sentencia de INSERT que añada un nuevo médico (la sentencia a ejecutar no tiene que dar error). Un ejemplo posible sería el que se muestra seguidamente:

T1	T3
SELECT * FROM medico	
	INSERT INTO medico VALUES (876, 'Ana Royo', 'Pediatría', 'Murillo', 'Sevilla', '578999111', 2700, '56111676')
INSERT INTO paciente VALUES (123, 'Vicente Hijos', 'Alcalá 10', 'Madrid', '46789999')	
	COMMIT
SELECT * FROM medico	
COMMIT	

En este caso, la segunda SELECT de T1 recupera las dos filas de la primera SELECT, más una fila extra (el fantasma) que corresponde con el nuevo médico que inserta la transacción T3.

Glosario

ACID *f pl* Acrónimo formado por las palabras *atomicity*, *consistency*, *isolation* y *definitivity* (atomicidad, consistencia, aislamiento y definitividad) que indica las propiedades que toda transacción debe cumplir.

almacén de datos *m* Bases de datos orientadas a áreas de interés de la empresa que integran datos de distintas fuentes con información histórica y no volátil y que tienen como objetivo principal hacer de apoyo en la toma de decisiones.

BD *f* Sigla correspondiente a base de datos.

base de datos operacional *f* Base de datos destinada a gestionar el día a día de una organización, es decir, almacena la información en lo referente a la operativa diaria de una institución.

cancelación de una transacción *f* Finalización de una transacción sin que se confirmen las actualizaciones hechas en la BD.

clave subrogada *f* Identificador único de una tabla construido a partir de una secuencia numérica autogenerada y que no se deriva de los datos de la aplicación.

common table expression *f* Tipo de consultas que utilizan la cláusula de SQL `WITH` con el fin de simplificar y facilitar la construcción de consultas complejas.

confirmación de una transacción *f* Finalización de una transacción que causa que los cambios realizados pasen a ser definitivos en la BD.

control de concurrencia *m* Conjunto de técnicas que utiliza un SGBD para evitar que se produzcan interferencias entre transacciones que se ejecutan de forma concurrente.

CTE *f* Véase ***common table expression***.

data warehouse *m* Véase **almacén de datos**.

dimensión *f* Punto de vista utilizado en el análisis de un cierto hecho.

ETL *m* Véase ***extract, transform and load***.

extract, transform and load *m* Conjunto de procesos en entornos *data warehouse* que se encargan de la extracción de datos procedentes de múltiples orígenes, de la transformación de estos para adecuarlos a las nuevas estructuras, y de su carga final en el almacén de datos para su consumo.

función analítica *f* Funcionalidad proporcionada por SQL para la realización de cálculos dentro de un contexto de forma que una fila vea y utilice datos más allá de aquellos asociados a dicha fila.

hecho *m* Objeto de análisis.

horario *m* La ejecución concurrente de un conjunto de transacciones (en las que se preserva el orden de las operaciones dentro de cada transacción) recibe el nombre de horario o historia.

interferencia *f* Comportamiento anómalo que puede producir el acceso concurrente de diversos usuarios a la BD, si no se toman las precauciones adecuadas y que pone en peligro la integridad de la misma o hace que llegue información errónea a los usuarios.

nivel de aislamiento *m* Grado de protección que ofrece el SGBD a una transacción según los tipos de interferencias de los que la protege.

nivel de concurrencia *m* Grado de aprovechamiento de los recursos de proceso disponibles según el solapamiento de ejecución de las transacciones que acceden de forma concurrente a la BD y que consiguen confirmar.

operación E/S *f* Véase **operación de entrada/salida**.

métrica *f* Dato numérico asociado a un acontecimiento que queremos analizar.

OLAP *m* Siglas que hacen referencia a las herramientas de análisis, normalmente multidimensional. Categoría de tecnología de software que permite a los analistas, gestores y ejecutivos mejorar su conocimiento de los datos mediante el acceso rápido, consistente e interac-

tivo a una amplia variedad de posibles vistas de información que ha sido transformada desde los datos operacionales para reflejar la dimensionalidad real de la empresa como la entiende el usuario (The OLAP Council).

OLTP *m* Siglas que hacen referencia a sistemas operacionales, que ayudan en el día a día de nuestra empresa.

operación de entrada/salida *f* Proceso que permite el transporte de páginas entre la memoria interna y la memoria externa del ordenador. En cada operación de entrada/salida (E/S) se transfiere un cierto número de páginas. En general, para simplificar los cálculos de coste, se asume que en cada operación de E/S se transfiere una página.

página *f* Unidad mínima de acceso y de transferencia de datos entre la memoria interna (o principal o intermedia) del ordenador y la memoria externa (no volátil) que contiene los ficheros de una base de datos. El SO de la máquina lleva a cabo esta transferencia y pasa la página al SGBD para que gestione su contenido. La página también es la estructura que permite al SGBD organizar los datos de una base de datos. En el área de SO la página recibe el nombre de bloque.

partición *f* Sistema de almacenamiento que permite distribuir los datos de una tabla en distintos espacios físicos para aumentar la eficiencia del sistema.

SGBD *m* Véase **sistema de gestión de bases de datos**.

sistema de gestión de bases de datos *m* Software que gestiona y controla bases de datos. Sus funciones principales son las de facilitar el uso simultáneo a muchos usuarios de distintos tipos, independizar al usuario de las estructuras físicas que implementan la base de datos y mantener la integridad de los datos.

sistema operacional *m* Sistema que ayuda en las operaciones diarias de negocio de una organización.

transacción *f* Conjunto de operaciones de lectura y/o actualización de la base de datos que acaba confirmando o cancelando los cambios que se han llevado a cabo.

Bibliografía

Adamson, C. (2010). *Star Schema: The Complete Reference*. McGraw-Hill.

Bernstein, P. A.; Newcomer, E. (2009). *Principles of Transaction Processing* (2.ª ed.). Burlington (Massachusetts): Morgan Kaufmann Publishers.

Chauhan, C. (2015). *PostgreSQL Cookbook*. Packt Publishing.

Elmasri, R.; Navathe, S. B. (2007). *Sistemas de bases de datos. Conceptos fundamentales* (5.ª ed.). Madrid: Addison-Wesley Iberoamericana.

Kimball, R.; Ross, M. (2013). *The Data Warehouse Toolkit* (3.ª ed.). John Wiley & Sons, Inc.

Liu, L.; Özsu, M. T. (eds.) (2009). *Encyclopedia of Database Systems*. Springer.

Morton, K; Osborne, K.; Sands, R.; Shamsudden, R; Still, J. (2013). *Pro Oracle SQL* (2.ª ed.). Apress.

Obe, R. O.; Hsu, L. S. (2014). *PostgreSQL: Up and Running* (2.ª ed.). O'Reilly Media, Inc.

Oracle. Manuales accesibles en línea desde la web http://www.oracle.com/technetwork/indexes/documentation/index.html.

PostgreSQL. Manuales accesibles en línea desde la web http://www.postgresql.org/docs/.

Prigmore, M. (2008). *An introduction to databases with web applications*. Pearson.

Shahzad, A.; Fayyaz, A.; Ahmed, I. (2015). *PostgreSQL Developer's Guide*. Packt Publishing.

Anexos: complementos de SQL – anotaciones para Oracle

En los siguientes anexos se presentan las anotaciones en SQL pertenecientes al SGBD Oracle, en las que se enfatizan las diferencias entre este SGBD y PostgreSQL. Las sentencias SQL proporcionadas, así como las explicaciones, están basadas en los ejemplos propuestos respectivamente en cada una de las secciones de este módulo didáctico.

Anexo 1. Claves subrogadas

1) Secuencias

La definición de secuencias en Oracle se realiza de la siguiente manera:

```
CREATE SEQUENCE sequence_name
       [ INCREMENT BY integer ]
       [ START WITH integer ]
       [ MAXVALUE integer | NOMAXVALUE ]
       [ MINVALUE integer | NOMINVALUE ]
       [ CYCLE | NOCYCLE ]
       [ CACHE # | NOCACHE ]
       [ ORDER | NOORDER ]
```

Como ejemplo, definiremos la secuencia que la tabla Asignatura utilizará:

```
CREATE SEQUENCE seq_asignatura_key INCREMENT BY 1 START WITH 1 NOCYCLE
```

Para hacer uso de la secuencia en Oracle, las secuencias disponen de la función `nextval`. Esta función es similar a la mostrada en PostgreSQL, con la diferencia de que no dispone de parámetros y que esta se llama con el nombre de la secuencia al inicio:

Nota

La explicación de los parámetros utilizados en la creación de una secuencia se puede consultar en la siguiente URL de Oracle 11g:

http://docs.oracle.com/cd/B28359_01/server.111/b28286/statements_6015.htm#SQLRF01314

```
SELECT seq_asignatura_key.nextval FROM dual
```

A la hora de insertar una fila nueva en la tabla Asignatura, podemos hacer una llamada a la función como parte de la sentencia `INSERT`:

```
INSERT INTO asignatura (asignatura_key, cod_asignatura, nom_asignatura)
VALUES (seq_asignatura_key.nextval, 'UOC-31238', 'Cálculo Numérico')
```

Si queremos automatizar esta solución para evitar utilizar el nombre de la secuencia en cada sentencia `INSERT`, podemos utilizar un disparador:

```
CREATE TRIGGER tg_asignatura_key
    BEFORE INSERT ON asignatura
    FOR EACH ROW
  BEGIN
    SELECT seq_asignatura_key.nextval INTO :new.asignatura_key
      FROM dual;
  END
;
INSERT INTO asignatura (cod_asignatura, nom_asignatura)
VALUES ('UOC-31299', 'Programación Java');
```

2) Uso de *IDENTITY*

En versiones de Oracle 11g e inferiores, no existe un tipo de dato o cláusula específica que permita generar valores numéricos autoincrementales. En su versión 12c, Oracle ha implementado esta funcionalidad al incorporar la cláusula `IDENTITY`, que tiene un funcionamiento similar al tipo de dato `serial` visto en PostgreSQL. Esta cláusula se asocia a columnas con tipos de datos numéricos a la hora de especificar las sentencias de creación de tablas. La sintaxis de esta cláusula es la siguiente:

Nota

Para más información acerca de esta cláusula en Oracle 12c, recomendamos visitar la documentación en línea, a la que se puede acceder a través de esta URL:

http://docs.oracle.com/database/121/SQLRF/toc.htm
http://docs.oracle.com/database/121/SQLRF/toc.htm

```
CREATE TABLE table_name (
   col_name numeric_type GENERATED [ALWAYS | BY DEFAULT] AS IDENTITY,
       ...
);
```

Anexo 2. *Common table expression*

1) Construcción de CTE

En el caso de Oracle, la posibilidad de crear consultas CTE se ha introducido en la versión Oracle 9.2, y la posibilidad de crear consultas recursivas desde la versión de Oracle 11g Release 2. La sintaxis de creación de CTE en Oracle es muy similar a la proporcionada para PostgreSQL, en la que se añaden funcionalidades específicas de este SGBD.

```
WITH alias_1 [ ( c_alias_1, c_alias_2, ... ] ) AS (
    query_1
)
[search_clause]
[cycle_clause]
, ...
     alias_n [ ( c_alias_1, c_alias_2, ... ] ) AS (
    query_n
)
[search_clause]
[cycle_clause]

SELECT ...
  FROM ...
 WHERE ...

[hierarchical_query_clause]

GROUP BY ...
ORDER BY ...
```

Por un lado, podemos ver que la palabra `RECURSIVE` no es necesaria en Oracle. Por otro lado, Oracle añade una funcionalidad especial para cada consulta auxiliar: `search_clause` y `cycle_clause`, que se pueden ver a continuación.

Nota

Para más información acerca de las CTE en Oracle 11g, visitad el enlace siguiente:

http://docs.oracle.com/cd/E11882_01/server.112/e41084/statements_10002.htm#SQLRF01702

La cláusula `search_clause` se utiliza para especificar ordenación en las filas:

```
SEARCH
      { DEPTH FIRST BY c_alias [, c_alias]...
          [ ASC | DESC ]
          [ NULLS FIRST | NULLS LAST ]
       | BREADTH FIRST BY c_alias [, c_alias]...
          [ ASC | DESC ]
          [ NULLS FIRST | NULLS LAST ]
      }
      SET ordering_column
```

- La ordenación se establece especificando las columnas (lista de `c_alias`) a partir de la cláusula `FIRST BY`.

- La lista de columnas (`c_alias`) deben de ser nombres de columnas de la lista de la consulta auxiliar (`c_alias_1, c_alias_2`, etc.).

- Si especificamos `BREADTH FIRST BY`, las filas hermanas se devolverán primero antes que las filas hijas. Al contrario, si se especifica `DEPTH FIRST BY`, las filas hijas se devolverán primero antes que las filas hermanas.

- `SET ordering_column` se utiliza para especificar el nombre del orden establecido por la cláusula `SEARCH`, y que puede ser utilizado por la consulta principal para devolver los datos en ese mismo orden.

La cláusula `cycle_clause` se utiliza para especificar ciclos de valores cuando se aplica recursividad. Básicamente, lo que se está haciendo es crear una nueva columna que puede ser referenciada en la consulta principal y que nos permite detectar si ciertos valores han sido repetidos o no dentro de las filas ancestros dentro de la jerarquía:

```
CYCLE c_alias [, c_alias]...
    SET cycle_mark_c_alias TO cycle_value
    DEFAULT no_cycle_value
```

- La lista de columnas (`c_alias`) deben ser nombres de columnas de la lista de la consulta auxiliar (`c_alias_1, c_alias_2`, etc.).

- Se tienen que especificar dos valores: el valor `cycle_value`, que es el valor que se asigna a la columna generada si se ha encontrado un ciclo, y el valor `no_cycle_value`, que es el valor por defecto cuando no se ha encontrado un ciclo. Ambos valores han de ser una secuencia de caracteres alfanuméricos de al menos 1 carácter.

- En el caso de que se encuentre un ciclo para una fila en concreto, el proceso de recursión para el cálculo de dicho ciclo se para en dicha fila, es decir, no seguirá buscando. Sí continuará para las filas que todavía no han encontrado un ciclo.

Ejemplo de consulta CTE con `SEARCH` y `CYCLE`

Utilizando la tabla de empleados, queremos obtener el listado de estos (nombre), junto al identificador de supervisor y ciudad. Para explicar el uso de `SEARCH` y `CYCLE`, generaremos la jerarquía en el orden de aparición en la jerarquía alfabéticamente mediante `SEARCH`, y marcaremos al empleado en el caso de que este viva en la misma ciudad que alguno de sus supervisores mediante `CYCLE`. La consulta que nos proporciona esta información es la siguiente (ved las partes resaltadas en negrita).

```
WITH jerarquia (id_empleado,
                nombre,
                id_supervisor,
                nivel_jerarquia,
                ciudad ) AS (
    SELECT
        id_empleado,
        nombre,
        id_supervisor,
        0 nivel_jerarquia,
        ciudad
    FROM  empleado
    WHERE id_supervisor IS NULL
    UNION ALL
    SELECT
        e.id_empleado,
        e.nombre,
        e.id_supervisor,
        d.nivel_jerarquia + 1 nivel_jerarquia,
        e.ciudad
    FROM  jerarquia d, empleado e
    WHERE d.id_empleado = e.id_supervisor
)
   SEARCH DEPTH FIRST BY id_supervisor, nombre SET orden_1
   CYCLE CIUDAD SET es_ciclo TO 'Y' DEFAULT 'N'
SELECT
    id_empleado ,
    lpad ('|- ', 3 * nivel_jerarquia ) || nombre AS empleado,
    id_supervisor,
    ciudad,
    es_ciclo
FROM  jerarquia
ORDER BY orden_1
```

Los resultados de esta consulta son los siguientes. Podemos ver que hemos intentado la jerarquía capturando el nivel en el que se encuentra cada empleado, y hemos presentado los datos de forma que primero mostramos las filas hijas y luego las filas hermanas (`DEPTH FIRST BY`) ordenadas alfabéticamente. Con esto lo que hacemos es realizar un recorrido en profundidad de un árbol. Además, si nos fijamos en los empleados Manuel Bertrán y Elena Rodríguez, vemos que se ha indicado `es_ciclo = 'Y'`. Esto significa que estos empleados viven en la misma ciudad que alguno de los supervisores de su jerarquía. En el caso de Manuel Bertrán, su supervisor José María Llopis vive en la misma ciudad, y en el caso de Elena Rodríguez es la supervisora Victoria Suárez la que vive en la misma ciudad.

Id Empleado	Nombre	Id Supervisor	Ciudad	Es Ciclo
10	Victoria Setan		Casteldefels	N
9	|- José María Llopis	10	Barcelona	N
11	|- Manuel Bertrán	9	Barcelona	Y
8	|- Víctor Anllada	10	Lleida	N
7	|- Victoria Suarez	10	Tarragona	N
4	|- Alejandra Martínez	7	Barcelona	N
6	|- Fernando Nadal	4	Viladecans	N
5	|- Marina Rodríguez	4	Vilanova	N
1	|- Manuel Vázquez	7	Barcelona	N
2	|- Elena Rodríguez	1	Tarragona	Y
3	|- José Pérez	1	Girona	N

2) Consultas jerárquicas

Oracle permite realizar consultas sobre datos jerárquicos sin utilizar consultas CTE. A estas consultas, Oracle las ha denominado **consultas jerárquicas**, y para su implementación, ha optado por la definición de la cláusula `CONNECT BY`, cuya sintaxis se puede ver a continuación:

Consultas jerárquicas

Para más información acerca de las consultas jerárquicas en Oracle 11g, visitad el enlace siguiente:

http://docs.oracle.com/cd/E11882_01/server.112/e41084/queries003.htm#SQLRF52335

```
CONNECT BY [NOCYCLE] condition [AND condition]... [START WITH condition]
| START WITH condition CONNECT BY [NOCYCLE] condition [AND condition]...
```

Si bien es una funcionalidad interesante para realizar consultas, esta está fuera del alcance de éste módulo, y por lo tanto no se profundizará en el tema.

Ejemplo de consulta jerárquica

La siguiente consulta obtiene la misma información que la consulta generada previamente con `SEARCH` y `CYCLE`, pero utilizando `CONNECTBY`.

```
SELECT
    id_empleado,
    LPAD ('|- ', 3 * (TO_NUMBER(LEVEL)- 1)) || nombre AS jerarquia,
    id_supervisor,
    ciudad
FROM
    EMPLEADO START WITH id_empleado = 10
CONNECT BY PRIOR id_empleado = id_supervisor
ORDER SIBLINGS BY nombre
```

Id Empleado	Nombre	Id Supervisor	Ciudad
10	Victoria Setan		Casteldefels
9	|- José María Llopis	10	Barcelona
11	|- Manuel Bertrán	9	Barcelona
8	|- Víctor Anllada	10	Lleida
7	|- Victoria Suarez	10	Tarragona
4	|- Alejandra Martínez	7	Barcelona
6	|- Fernando Nadal	4	Viladecans
5	|- Marina Rodríguez	4	Vilanova
1	|- Manuel Vázquez	7	Barcelona
2	|- Elena Rodríguez	1	Tarragona
3	|- José Pérez	1	Girona

Anexo 3. Funciones analíticas

1) Llamadas a funciones analíticas

La llamada a funciones analíticas en Oracle es muy similar a la que se ha propuesto en PostgreSQL. Estas han sido añadidas en la versión de Oracle 8.i Release 2. La sintaxis es la siguiente:

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

En esta definición, `arguments` representa cualquier expresión que no contenga una llamada a una función analítica: podría tratarse de una columna de una tabla, una función de agregación, una constante o un cálculo, entre otros.

La sintaxis de `analytic_clause` es la siguiente:

```
[ PARTITION BY { expr[, expr ]... | ( expr[, expr ]... ) }]

[ ORDER [ SIBLINGS ] BY
     { expr | position | c_alias }
     [ ASC | DESC ]
     [ NULLS FIRST | NULLS LAST ]
       [, { expr | position | c_alias }
     [ ASC | DESC ]
     [ NULLS FIRST | NULLS LAST ]
     [ frame_clause ]
]
```

en la que `frame_clause`, que permite definir el marco de trabajo, se define como:

```
{ ROWS | RANGE }
{  BETWEEN
    { UNBOUNDED PRECEDING | CURRENT ROW | offset { PRECEDING | FOLLOWING } }
   AND
    { UNBOUNDED FOLLOWING | CURRENT ROW | offset { PRECEDING | FOLLOWING } }

 | UNBOUNDED PRECEDING

 | CURRENT ROW

 | offset PRECEDING
}
```

A diferencia de PostgreSQL, Oracle tiene las siguientes particularidades:

a) Permite especificar en una misma consulta diferentes funciones analíticas con diferentes ventanas mediante `PARTITION BY`, pero no permite el uso de la cláusula `WINDOW` como en PostgreSQL.

b) La cláusula de definición del marco obliga a definir `ORDER BY`, es decir, no se puede definir un marco sin definir `ORDER BY` como parte de la función analítica.

c) La cláusula `ORDER BY` permite solamente una expresión en el caso de que se use `RANGE` y una expresión con *offset* para determinar los límites.

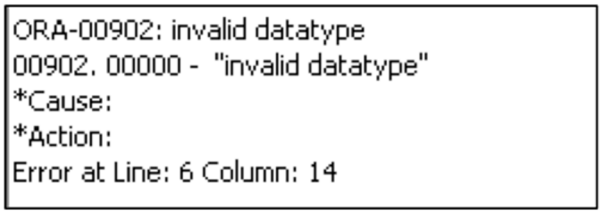
d) *Offset* solamente puede tomar valores de tipo numérico o un intervalo.

e) En el caso de que se use una expresión con *offset*, el tipo de dato de la expresión usada en `ORDER BY` debe ser numérica o de tipo fecha.

Como ejemplo, la siguiente consulta no sería válida porque el tipo de dato de *nombre_apellidos* es una cadena de caracteres.

```
SELECT
  ciudad,
  nombre_apellidos,
  num_asignaturas,
  SUM(num_asignaturas) OVER (PARTITION BY ciudad
    ORDER BY nombre_apellidos ASC
       RANGE BETWEEN UNBOUNDED PRECEDING AND 1 FOLLOWING) AS suma
FROM
  alumno
ORDER BY
  ciudad ASC,
  nombre_apellidos ASC
```

Figura 4. Error que produce Oracle si el tipo de dato en `ORDER BY` es diferente a una fecha o número.



f) En el caso de que se omita la definición del marco, la opción seleccionada por Oracle es `RANGE UNBOUNDED PRECEDING`, que es equivalente a especificar `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW`.

g) Al contrario que PostgreSQL, el límite inferior del marco sí puede tomar un valor `UNBOUNDED FOLLOWING`, pero obliga a que el límite superior sea también `UNBOUNDED FOLLOWING`. A su vez, el límite superior sí puede tomar un valor `UNBOUNDED PRECEDING`, pero el límite inferior también ha de ser definido como `UNBOUNDED PRECEDING`.

2) Tipos de funciones analíticas

Row number

La función `row_number` en Oracle no acepta parámetros.

Nota

Consultad el siguiente vínculo para obtener una descripción de todas las funciones analíticas presentes en Oracle 11g:
http://docs.oracle.com/cd/E11882_01/server.112/e41084/functions004.htm#SQLRF06174

```
row_number()
```

Rank

La función `rank` en Oracle no acepta parámetros.

```
rank()
```

Dense rank

La función `dense_rank` en Oracle no acepta parámetros.

```
dense_rank()
```

Lag

La función `lag` en Oracle acepta los siguientes parámetros:

```
lag ( expression [{RESPECT | IGNORE} NULLS] [, offset] [, default] );
```

- `expression`: cualquier valor a evaluar excepto funciones analíticas (por ejemplo, una columna de una tabla, una función escalar, etc.).
- `offset` (opcional): indica la posición de la fila previa a la que se va a acceder desde la fila actual en la partición. Por ejemplo, un valor de 3 indica que se va a acceder a la tercera fila previa a la fila actual. Si se omite, por defecto se asigna un valor 1 (la fila anterior).
- `default` (opcional): el valor por defecto a asignar en el caso de que la fila a acceder esté fuera de los límites permitidos. Si se omite, por defecto se asigna un valor `NULL`.

En el caso de Oracle, existe una funcionalidad que permite incluir o eliminar los valores nulos de `expression` del cálculo de la función. Esto se indica mediante las cláusulas `RESPECT NULLS` o `IGNORE NULLS` justo después de la expresión a evaluar. Por defecto, Oracle asume que se especifica `RESPECT NULLS` en el caso de omisión.

Lead

La función `lead` en Oracle acepta los siguientes parámetros:

```
lead ( expression [{RESPECT | IGNORE} NULLS] [, offset] [, default] );
```

- `expression`: cualquier valor a evaluar excepto funciones analíticas (por ejemplo, una columna de una tabla, una función escalar, etc.).
- `offset` (opcional): indica la posición de la fila posterior a la que se va a acceder desde la fila actual en la partición. Por ejemplo, un valor de 3 indica que se va a acceder a la tercera fila posterior a la fila actual. Si se omite, por defecto se asigna un valor 1 (la fila siguiente).
- `default` (opcional): el valor por defecto a asignar en el caso de que la fila a acceder esté fuera de los límites permitidos. Si se omite, por defecto se asigna un valor `NULL`.

En el caso de Oracle, y al igual que sucede con `lag`, es posible incluir o eliminar los valores nulos de `expression` del cálculo de la función mediante las cláusulas `RESPECT NULLS` o `IGNORE NULLS` justo después de la expresión a evaluar. Por defecto, Oracle asume que se especifica `RESPECT NULLS` en el caso de omisión.

First value

La función `first_value` en Oracle acepta los siguientes parámetros:

```
first_value ( expression [{RESPECT | IGNORE} NULLS] );
```

- `expression`: cualquier valor a evaluar excepto funciones analíticas (por ejemplo, una columna de una tabla, una función escalar, etc.).

En el caso de Oracle, y al igual que sucede con otras funciones analíticas, es posible incluir o eliminar los valores nulos de `expression` del cálculo de la función mediante las cláusulas `RESPECT NULLS` o `IGNORE NULLS` justo después de la expresión a evaluar. Por defecto, Oracle asume que se especifica `RESPECT NULLS` en el caso de omisión.

Last value

La función `last_value` en Oracle acepta los siguientes parámetros:

```
last_value ( expression [{RESPECT | IGNORE} NULLS] );
```

- `expression`: cualquier valor a evaluar excepto funciones analíticas (por ejemplo, una columna de una tabla, una función escalar, etc.).

En el caso de Oracle, y al igual que sucede con otras funciones analíticas, es posible incluir o eliminar los valores nulos de `expression` del cálculo de la función mediante las cláusulas `RESPECT NULLS` o `IGNORE NULLS` justo después de la expresión a evaluar. Por defecto, Oracle asume que se especifica `RESPECT NULLS` en el caso de omisión.