

# Tema 5 Implementación de Bases de Datos

David Moreno Lumbreras & Daniela Patricia Feversani

GSyC, EIF. URJC.

Laboratorio de Bases de Datos (BBDD)

Curso 24-25





(cc) 2023 - David Moreno Lumbreras & Daniela Patricia Feversani  
Algunos derechos reservados. Este trabajo se entrega bajo la licencia  
Creative Commons Reconocimiento - Compartirlgual  
(by-sa). Para obtener la licencia completa, véase  
<https://creativecommons.org/licenses/by-sa/3.0/es/>.

5.1 Introducción  
○○○

5.2 Traducción  
○○○○○○○○○

5.3 Organización  
○○○○○○○○○○○○○

5.4 Redundancia  
○○○○○○○○○○○○○

5.5 Seguridad  
○○○○○

5.6 Rendimiento  
○○○○

5.7 Transacciones  
○○○

Referencias  
○

# Contenidos

## 5.1 Introducción

# Implementación de Bases de Datos

- ▶ La implementación de una base de datos es un proceso clave para la correcta ejecución de la misma.
- ▶ Proceso que consiste en traducir el modelo conceptual de la base de datos a un Sistema de Gestión de Bases de Datos (DBMS o RDBMS).
- ▶ La forma en la cual se implementa la base de datos tiene un impacto directo en la **eficiencia y confiabilidad** de los sistemas.
- ▶ Para lograr es necesario abarcar los siguientes aspectos:
  - ▶ Traducción del Modelo a un DBMS.
  - ▶ Organización de Archivos e Índices.
  - ▶ Introducción de Redundancia Controlada.
  - ▶ Seguridad en las Bases de Datos.
  - ▶ Transacciones en Bases de Datos.

# Aspectos Clave de la Implementación de Bases de Datos

- ▶ **Traducción del Modelo a un DBMS:** Convertir nuestro modelo conceptual en una estructura que un Sistema de Gestión de Bases de Datos pueda entender y gestionar.
- ▶ **Organización de Archivos e Índices:** Diseñar la estructura de almacenamiento y los índices para optimizar la recuperación de datos y mejorar el rendimiento.
- ▶ **Introducción de Redundancia Controlada:** Estrategia para manejar la redundancia de datos de manera eficiente, equilibrando la coherencia y el rendimiento.
- ▶ **Seguridad en las Bases de Datos:** Establecer medidas de control de acceso y roles para garantizar la confidencialidad e integridad de los datos.
- ▶ **Transacciones en Bases de Datos:** Comprender cómo las transacciones aseguran la consistencia de los datos y mantienen la integridad en operaciones complejas.

## 5.2 Traducción del Modelo a un Sistema de Gestión de Bases de Datos Relacionales (RDBMS)

# Elección del Software de la Base de Datos

- ▶ La elección del Sistema de Gestión de Bases de Datos (DBMS) es la primera decisión crucial a la hora de implementar una Base de Datos.
- ▶ El DBMS gestiona la base de datos y que permite a los usuarios interactuar con ella.
- ▶ Existen diferentes aspectos y conceptos que se deben conocer a la hora de elegir un DBMS.
  - ▶ Componentes y características del DBMS.
  - ▶ Escalabilidad.
  - ▶ Tipo de DBMS.
  - ▶ Otros aspectos: Seguridad, Soporte, Costos o Integración.



# Componentes de un DBMS

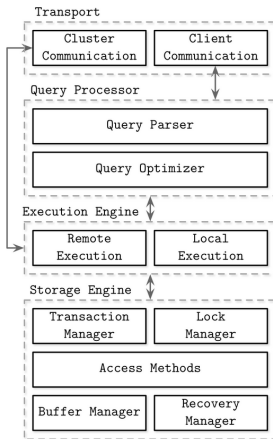
- ▶ **Transporte:** Es el responsable de transferir eficientemente los datos entre la aplicación y la base de datos.
  - ▶ En sistemas complejos (distribuidos) se encarga de coordinar las diferentes partes en las que puede estar ubicada la base de datos (nodos).
- ▶ **Procesador de consultas:** Asistente que permite al ordenador entender, verificar y realizar operaciones sobre la BD. Esta formado por:
  - ▶ *Parser:* interpreta y valida las consultas, haciendo además control de acceso.
  - ▶ *Optimizador:* Analiza la consulta y elige el mejor *plan de ejecución*, en función de varios criterios (Estadísticas, ubicación de los datos en disco, presencia de índices, etc).
- ▶ **Motor de ejecución:** Realiza la consulta según el plan elegido y recoge los resultados.
- ▶ **Motor de almacenamiento:** Organiza y almacena físicamente los datos en la base de datos. Decide dónde y cómo se guardan los datos en el disco o memoria y cómo se accede a ellos.
  - ▶ Especialmente en los DBMS relacionales, existen diferentes motores de almacenamiento para un mismo DBMS. Cada motor de almacenamiento tiene sus propias características y ventajas.

# Motores de Almacenamiento

- ▶ Por ejemplo, en MySQL podemos seleccionar entre diferentes *motores de almacenamiento* para el servidor <sup>1</sup>:
  - ▶ **InnoDB**: Motor por defecto desde MySQL 5.5.5. Ofrece soporte transaccional, ACID y MVCC (Control de Concurrencia Multi-Version).
  - ▶ **NDB (Oracle)**: Diseñado para configuraciones en cluster.
  - ▶ **XtraDB (Percona)**: Reemplazo *drop-in* para InnoDB en Percona Server y MariaDB (hasta MariaDB 10.2). Ofrece soporte ACID y MVCC.
  - ▶ **Aria (MariaDB)**: Un reemplazo más ligero (sin soporte ACID) para InnoDB.
  - ▶ **TokuDB (TokuTek)**: Proporciona soporte ACID y MVCC, utiliza índices mediante árboles fractales, es escalable y está preparado para SSD. Incluido en Percona Server y MariaDB.
- ▶ En PostgreSQL no hay variedad de motores independientes, pero su manejador de almacenamiento incorporado es altamente configurable y ofrece un conjunto de características sólidas para satisfacer un gran abanico de necesidades.

<sup>1</sup>[https://en.wikipedia.org/wiki/Comparison\\_of\\_MySQL\\_database\\_engines](https://en.wikipedia.org/wiki/Comparison_of_MySQL_database_engines)

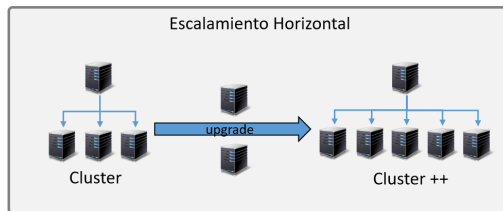
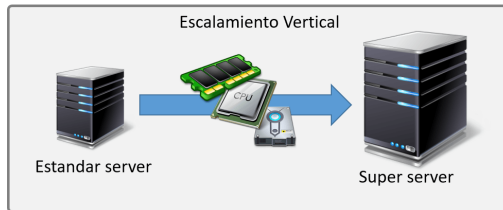
# Esquema de Arquitectura Interna de un DBMS



# Escalabilidad

- ▶ Capacidad del DBMS de manejar un aumento en la carga de trabajo, ya sea en términos de datos, tráfico de consultas o usuarios concurrentes.
- ▶ Aspecto crucial para la implementación de una base de datos → Asegurar que el sistema pueda crecer y adaptarse a demandas mayores sin ver afectado su rendimiento.
- ▶ Se consideran dos tipos principales de escalabilidad:
  - ▶ **Escalabilidad Vertical** (*scaling up o Escalar hacia arriba*): Migrar la base de datos a otro sistema más grande y potente (problemas de tiempo para completar el proceso y, en consecuencia, disponibilidad).
  - ▶ **Escalabilidad Horizontal** (*scaling out o Escalar hacia afuera*): Ejecutar múltiples instancias de la misma base de datos en nuevos nodos o servidores, distribuyendo la carga de trabajo, para mejorar rendimiento y capacidad.

# Escalabilidad



# Tipos de Sistemas de Gestión de Bases de Datos

- ▶ Existen diferentes tipos de DBMS, con diferentes características y ventajas.
  - ▶ **Sistemas de Gestión de Bases de Datos Relacionales (RDBMS):** Emplean un modelo relacional para organizar datos en tablas y relaciones entre ellas. Se basan en el lenguaje SQL.
  - ▶ **Sistemas de Gestión de Bases de Datos No Relacionales (NoSQL):** Diseñados para manejar grandes volúmenes de datos no estructurados o semiestructurados. Permiten una mayor flexibilidad en la estructura de datos.
  - ▶ **Sistemas de Gestión de Bases de Datos en Memoria:** Almacenan y recuperan datos directamente desde la memoria principal, en lugar de utilizar almacenamiento en disco. Este hecho provoca que el acceso a los datos sea más rápido.

# Comparación de Tipos de DBMS

Tipo de DBMS	Ventajas	Inconvenientes	Ejemplos
Relacional (RDBMS)	<ul style="list-style-type: none"> <li>▶ Modelo estructurado y consistente.</li> <li>▶ Soporte ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad) para transacciones.</li> </ul>	<ul style="list-style-type: none"> <li>▶ Menor flexibilidad en el esquema.</li> <li>▶ Rendimiento puede degradarse con grandes volúmenes de datos.</li> </ul>	<ul style="list-style-type: none"> <li>▶ MySQL (libre)</li> <li>▶ PostgreSQL (libre)</li> <li>▶ Microsoft SQL Server (propiedad)</li> <li>▶ Oracle Database (propiedad)</li> <li>▶ MariaDB (libre)</li> </ul>
NoSQL	<ul style="list-style-type: none"> <li>▶ Mayor flexibilidad de esquema.</li> <li>▶ Escalabilidad horizontal sencilla.</li> </ul>	<ul style="list-style-type: none"> <li>▶ Menos soporte para transacciones ACID.</li> <li>▶ Puede requerir habilidades específicas para consultas complejas.</li> </ul>	<ul style="list-style-type: none"> <li>▶ MongoDB (libre)</li> <li>▶ Cassandra (libre)</li> <li>▶ Redis (libre)</li> </ul>
En Memoria	<ul style="list-style-type: none"> <li>▶ Rápido acceso a datos en la memoria.</li> <li>▶ Adecuado para casos de baja latencia.</li> </ul>	<ul style="list-style-type: none"> <li>▶ Limitado por la capacidad de la memoria.</li> <li>▶ La persistencia de datos puede ser un desafío.</li> </ul>	<ul style="list-style-type: none"> <li>▶ Redis (libre)</li> <li>▶ Apache Ignite (libre)</li> <li>▶ Memcached (libre)</li> </ul>

## 5.3 Organización de Archivos e Índices



## Conceptos básicos

- ▶ En una consulta se extrae una pequeña parte de todos los registros, por lo que no es eficiente que el sistema tenga que leer cada registro y comprobar que el campo cumple lo que se está buscando.
- ▶ Este proceso se facilita mediante el diseño de estructuras adicionales que se asocian con los archivos (tabla).
- ▶ Se conoce como **Índice** y, tienen una analogía con el índice de un libro.
- ▶ Un índice es un objeto más de la base de datos. Está formado por una o más columnas y por un puntero a la fila de la tabla que representa, con la interesante característica de que las filas dentro del índice mantienen siempre un orden especificado por algún algoritmo.

# Tipos de Índices

- ▶ Existe dos tipos principales de índices:
  - ▶ **Índices Ordenados:** Basados en una disposición ordenada de los valores.
    - ▶ Organizan la información de manera ordenada (ascendente o descendente).
    - ▶ Mejoran la eficiencia en la búsqueda y recuperación de datos.
    - ▶ *B-Tree* es un tipo común de índice ordenado utilizado en bases de datos relacionales.
  - ▶ **Índices Asociativos o Hash:** Basados en una distribución uniforme de los valores a través de una serie de cajones. El valor asociado a cada cajón está determinado por una función (*Función de asociación o Hash function*).
    - ▶ Proporcionan acceso rápido a los datos mediante la creación de una relación directa entre la clave y su ubicación en la base de datos.
    - ▶ *Índices Hash* son un ejemplo de índices asociativos. Dan una búsqueda eficiente cuando se conoce la clave.

# Índices Ordenados

- ▶ Como norma general, *siempre* que creamos una **clave primaria** se crea un índice asociado a la misma. De esa forma, se suele garantizar (en la mayoría de formatos de fichero para almacenamiento de tablas) que las filas están ordenadas según dicha clave primaria.
- ▶ Además de las claves primarias, se pueden definir **índices secundarios** en una tabla, es decir, índices adicionales sobre el contenido de columnas que no son la clave primaria.
- ▶ En este caso, se suelen adoptar dos posibles estrategias para reconocer qué columna debemos indexar:
  - ▶ Seleccionar aquella columna o columnas que más se usan en operaciones **JOIN**, para buscar las correspondencias entre dos tablas.
  - ▶ Seleccionar aquella columna o columnas de la tabla que más se van a usar en una consulta para recuperar de la misma datos ordenados.

# Índices Ordenados

- ▶ Entonces, ¿por qué no crear sistemáticamente índices sobre varias columnas? Si siempre acelera...
- ▶ Ciertamente, pero la **creación de índices** tiene varios **costes**:
  - ▶ Coste de **espacio**: El índice creado ocupa espacio físico en disco.
  - ▶ Coste de **tiempo**: En tablas muy grandes, se puede tardar mucho tiempo en indexar los datos la primera vez. Hay que tener en cuenta que la tabla está siempre ordenada respecto a los valores de la clave primaria, por lo que el orden de las filas no se puede cambiar luego, aunque se generen índices secundarios.
  - ▶ Coste **computacional**: Dependiendo del tipo de índice generado, del algoritmo utilizado para crearlo y del tipo de dato de la columna(s), puede ser más o menos costoso computacionalmente crearlo.
- ▶ En la práctica, hay que elegir cuidadosamente y de manera muy razonada qué columnas van a tener índices secundarios en una tabla.
- ▶ Si empezamos a crear más índices de los estrictamente necesarios, el tamaño de esos índices en disco puede crecer rápidamente... incluso hasta hacerse más grandes que la propia tabla de datos (!!).

# Índices Ordenadas

- ▶ Evita crear índices sobre campos de una tabla que se actualicen con frecuencia.
- ▶ Evita crear índices sobre campos que sean de tipo *string* y de gran longitud.
  - ▶ Son siempre los más ineficientes y costosos, tanto cuando creamos el índice como cuando lo usamos. Un truco suele ser (si lo soporta nuestra BD) indexar la columna empezando por los caracteres del final (en orden inverso a como están guardados), sobre todo si los primeros caracteres son parecidos en muchos casos.
  - ▶ Un ejemplo sería un campo tipo **VARCHAR** que almacena URLs.

# Ejemplo: Índices Ordenados

id	nombre	salario	índice	nombre
1	Carlos	60000	1	Ana
2	Ana	50000	2	Carlos
3	Miguel	55000	3	Elena
4	Elena	48000	4	Miguel

- ▶ Si se crea una estructura de índices para el campo "id", estaremos ante un caso de *Índices primarios o índices con agrupación* dado que el orden de este campo es secuencial.
- ▶ Si se crea una estructura de índices para el campo "nombre", se está ante un caso de *Índices secundarios o índices sin agrupación*.

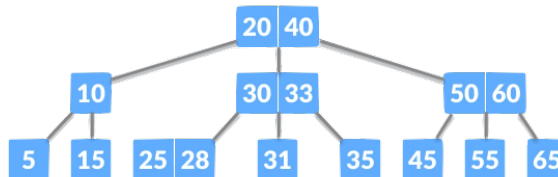
## Ejemplo: Índices Ordenados

- ▶ Si no se crea una estructura de índices, al realizar una consulta se recorren todos los registros buscando la coincidencia.
- ▶ Si se crea un índice a la columna nombres, se tendrá un diccionario ordenado de los nombres con el conjunto (Número\_Índice:Nombre).
- ▶ Se agilizará la búsqueda al buscar el nombre y, con el índice hacer a la posición en memoria del registro o registros en cuestión.

```
CREATE INDEX idx_nombre ON empleados(nombre);
```

# Índices Ordenados: B-Tree

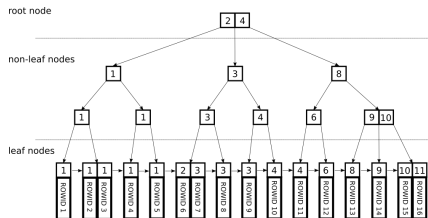
- ▶ En el caso de PostgreSQL, los índices ordenados por defecto son los denominados *B-Tree* o *Balanced Tree*.
- ▶ Estructura de datos en forma de árbol que organiza las claves de manera ordenada para facilitar la búsqueda, inserción y eliminación eficiente.
- ▶ Cada nodo del árbol (a excepción del raíz) tiene un número mínimo y máximo de claves y punteros, de tal forma que el árbol este equilibrado y se mantenga la eficiencia en las operaciones.
- ▶ El árbol se reorganiza cada vez que se inserte o se elimine una clave de tal forma que todas las hojas estén a la misma distancia de la raíz.





# Índices Ordenados: B-Tree

- ▶ Los nodos se agrupan de manera lógica en tres grupos.
- ▶ **Root node**: No tiene padres, está al comienzo del árbol (parte más alta).
- ▶ **Leaf nodes**: Están al final del árbol (parte más baja), no tienen hijos.
- ▶ **Internal nodes**: El resto de nodos intermedios, que conectan la raíz con los hijos. Normalmente, hay varios niveles.



# Índices Ordenados: B-Tree

- ▶ Esta estructura presenta una serie de ventajas a la hora de trabajar con los DBMS.
  - ▶ **Eficiencia de Búsqueda:** Al estar las claves organizadas en orden, la búsqueda se realiza de manera eficiente.
  - ▶ **Inserción y Eliminación Eficientes:** Esta estructura permite realizar operaciones de inserción y eliminación eficientemente, manteniendo el equilibrio del árbol.
  - ▶ **Adaptabilidad a cambios:** Son estructuras dinámicas que se adaptan correctamente a los procesos de inserción y eliminación de datos de manera frecuente sin comprometer de manera significativa el rendimiento.
- ▶ El uso de índices también presenta inconvenientes:
  - ▶ Los propios índices ocupan espacio físico.
  - ▶ Es cierto que los índices se autordenan, pero se realizan operaciones para realizar ese proceso de ordenación que provocan aumentos de tiempo a la hora de insertar o borrar registros debido a la actualización de los campos indexados.

## Caso de estudio: PostgreSQL

- ▶ Si no se indica explícitamente ninguna otra opción, PostgreSQL utilizará un B-Tree para crear un nuevo índice.
- ▶ Otros tipos de índices estándar en PostgreSQL.
  - ▶ Block range index (BRIN): Para tablas muy grandes, tratan un rango de páginas como una unidad. Más pequeños y fáciles de construir pero lentos en acceso y no soportan claves primarias.
  - ▶ Generalized Search Tree (GiST): Optimizado para FTS (Full-text Search), datos espaciales, científicos, no estructurados y jerárquicos. Guarda límites de almacenamiento no valores exactos indexados (*lossy index*).
  - ▶ Generalized Inverted Index (GIN): Para la FTS integrada y datos JSON binarios. Basado en GiST pero sin pérdidas. Más grande y actualizaciones más lentas que GiST.

## 5.4 Introducción a la Redundancia Controlada

# Desnormalización

- ▶ Introducción a propósito de redundancias en nuestro esquema, no acatando las directrices de normalización.
- ▶ Objetivo: Mejorar el rendimiento de consultas de acceso y recuperación de información.
- ▶ Riesgos:
  - ▶ Dificulta la implementación (e impone cambios sobre el modelo que “ya estaba acabado...”).
  - ▶ Al ir en contra de los principios de normalización, se sacrifica flexibilidad (se aumenta el acoplo entre datos de una tabla).
  - ▶ Puede *perjudicar mucho* el rendimiento de operaciones de inserción/borrado/actualización de datos (por no mencionar los problemas derivados del mantenimiento de integridad referencial y de datos).

# Desnormalización

- ▶ Comentamos solo algunos casos comunes (no hay reglas “universales”).
- ▶ Duplicar atributos que no sean clave en relaciones 1:\*, para reducir los JOIN.
- ▶ Duplicar claves foráneas en relaciones 1:\*, para reducir los JOIN.
- ▶ Duplicar atributos en relaciones \*: para reducir los JOIN.
- ▶ Introducción de grupos de datos repetidos (contra 3NF).
- ▶ Crear tablas de extracción de datos particulares para queries específicas (mejor crear vistas, si es necesario materializadas).
- ▶ Creación de vistas, Herencia y Particionado de tablas.

# Vistas

- Una **vista** representa el resultado de una consulta mediante un nombre, de forma que a partir de entonces se pueda acceder a dichos datos utilizando el nombre de la *vista* como si fuese una tabla (virtual).

```
CREATE VIEW order_details_view AS
SELECT o.orderinfo_id, o.date_placed, o.date_shipped, c.customer_id, c.fname,
c.lname, ol.item_id, i.description AS item_description, ol.quantity
FROM orderinfo o
JOIN customer c ON o.customer_id = c.customer_id
JOIN orderline ol ON o.orderinfo_id = ol.orderinfo_id
JOIN item i ON ol.item_id = i.item_id;
```

```
SELECT * FROM order_details_view;
```

# Vistas

- ▶ Aunque depende de la BD concreta que usemos, las vistas normalmente no se crean de verdad en disco, sino que son un *alias* en memoria para realizar la consulta que crea los datos de la vista.
- ▶ De este modo, hace más legibles consultas que siempre usan esa tabla virtual (si no, tendríamos que usar constantemente una subconsulta, quizá muy larga, que degrada la legibilidad de la consulta principal).
- ▶ No todas las vistas son *actualizables*.
  - ▶ De hecho, a veces se definen vistas sobre las columnas de una tabla para dejar otras columnas de esa tabla fuera de la vista para un perfil de usuario.
  - ▶ Así, ese perfil de usuario no “ve” esas columnas y las protegemos frente a modificaciones indebidas.



## Vistas materializadas

- ▶ En general, las vistas son *virtuales*, es decir, se tienen que generar cada vez que se mencionan en una consulta. Este proceso se llama **resolución de la vista**.
- ▶ El proceso se repite tantas veces como consultas incluyan esa vista, por lo que si las tablas involucradas son muchas, de gran tamaño, o la consulta implicada en la resolución de la vista es compleja puede añadir una sobrecarga de tiempo importante para resolverla.
- ▶ Una alternativa es lo que llamamos **vista materializada**: una vista cuya información se guarda temporalmente (en memoria o disco), de forma que solo se tiene que construir la primera vez que se usa.
- ▶ Una importante desventaja de las vistas materializadas es que, si son muy grandes, ocupan espacio en disco (a veces mucho) o en memoria, consumiendo recursos.

# Vistas materializadas: Ejemplo

```
-- Crear la vista materializada
CREATE MATERIALIZED VIEW item_stock_view AS
SELECT
    i.item_id,
    i.description,
    s.quantity AS stock_quantity
FROM
    item i
JOIN
    stock s ON i.item_id = s.item_id;

-- Crear un índice en la vista materializada para mejorar el rendimiento de las consultas
CREATE UNIQUE INDEX idx_item_stock_view ON item_stock_view(item_id);

SELECT * FROM item_stock_view;
```

# Tablas Heredadas

- ▶ PostgreSQL es el único software de BD hasta la fecha que ofrece **tablas heredadas**.
- ▶ Cuando se especifica que una tabla (*child*) hereda de otra (*parent*), la tabla *child* se crea con sus propias columnas, más todas las columnas de la tabla *parent*.
- ▶ Una aplicación de mucha utilidad es para el particionado de datos: cuando se consulta la tabla *parent*, PostgreSQL incluye automáticamente todas las filas en las tablas *child*.
- ▶ No se hereda la clave primaria, las claves foráneas, restricciones de unicidad o índices que se hayan definido en la tabla *parent*.

# Tablas Heredadas

```
-- Tabla base 'persona'
CREATE TABLE persona (
    id SERIAL PRIMARY KEY,
    nombre VARCHAR(50),
    edad INTEGER
);

-- Tabla hija 'estudiante' que hereda de 'persona'
CREATE TABLE estudiante (
    expediente INTEGER PRIMARY KEY,
    carrera VARCHAR(50)
) INHERITS (persona);

-- Tabla hija 'profesor' que también hereda de 'persona'
CREATE TABLE profesor (
    idprofesor INTEGER PRIMARY KEY,
    departamento VARCHAR(50)
) INHERITS (persona);
```

# Tablas Particionadas

- ▶ Tienen un uso similar al de las tablas heredadas, porque sirven para particionado de datos. Sin embargo, usan una sintaxis diferente. Existen desde PostgreSQL v10.
- ▶ Las tablas particionadas se declaran con la sintaxis:  
`CREATE TABLE ... PARTITION BY RANGE....`
- ▶ Los datos insertados en la tabla son redirigidos automáticamente a la partición correspondiente. En las tablas heredadas, hay que insertarlos manualmente en la tabla adecuada o tener un *trigger* que encamine los datos a la tabla heredada.

# Tablas Particionadas

```
-- Crear la tabla base 'ventas' particionada por rango de fechas
CREATE TABLE ventas (
    id SERIAL PRIMARY KEY,
    fecha_venta DATE,
    monto DECIMAL
) PARTITION BY RANGE (fecha_venta);

-- Crear dos particiones para trimestres diferentes
CREATE TABLE ventas_q1 PARTITION OF ventas
    FOR VALUES FROM ('2023-01-01') TO ('2023-03-31');

CREATE TABLE ventas_q2 PARTITION OF ventas
    FOR VALUES FROM ('2023-04-01') TO ('2023-06-30');
```

## 5.5 Seguridad y Control de Acceso

## Gestión de Permisos y Roles.

- ▶ La gestión de permisos y roles en DBMS es fundamental desde el punto de vista de la seguridad y la integridad de los datos.
- ▶ En PostgreSQL, existen herramientas que permiten la gestión de estos aspectos.
- ▶ **Roles:** Usuarios con permisos específicos. Algunos de los roles más comunes son:
  - ▶ **Superusuario:** Rol con todos los privilegios sobre la BD (Administrador).
  - ▶ **Usuario:** Rol con privilegios limitados y, que representa a los individuos que acceden a la BD.
  - ▶ **Rol de Aplicación:** Rol creado para una aplicación específica y que tiene permisos limitados.
- ▶ **Permisos:** Definen qué operaciones están permitadas para un rol específico. Los permisos pueden aplicarse a diferentes niveles:
  - ▶ **Base de Datos:** Acceder y manipular la base de datos en su conjunto.
  - ▶ **Esquema:** Acceder y manipular objetos dentro de un esquema específico.
  - ▶ **Tabla o Vista:** Leer, escribir o modificar datos en una tabla o vista concreta.
- ▶ <http://www.postgresqltutorial.com/postgresql-roles/>.



## Gestión de Permisos y Roles.

- ▶ Las cláusulas y sentencias más relevantes en PostgreSQL para la gestión de permisos y roles son:
  - ▶ Creación de Rol: `CREATE ROLE nombre_del_rol;`
  - ▶ Consultar Roles: `SELECT rolname FROM pg_roles;`
  - ▶ Asignar Permisos a un Rol:
    - `GRANT permiso_nombre ON esquema_nombre TO nombre_del_rol;`
  - ▶ Asignar Permisos a un rol para una tabla o vista:
    - ▶ `GRANT SELECT ON vista_nombre TO nombre_del_rol;`
    - ▶ `GRANT SELECT, INSERT, UPDATE, DELETE ON nombre_tabla TO nombre_del_rol;`
  - ▶ Revocar Permisos a un Rol:
    - `REVOKE permiso_nombre ON esquema_nombre FROM nombre_del_rol;`
  - ▶ Revocar Permisos a un rol para una tabla o vista:
    - `REVOKE SELECT ON vista_nombre FROM nombre_del_rol;`
  - ▶ Eliminar Rol: `DROP ROLE nombre_del_rol;`

# Principios de Seguridad

- ▶ En los DBMS, junto a aspectos como los roles y permisos, existen una serie de principios de seguridad que se deben cumplir para garantizar la integridad y la no vulnerabilidad de los datos.
- ▶ En la informática se considera que al menos son necesarios aplicar 4 principios básicos de seguridad informática para proteger toda la información de una organización. Estos 4 principios son:
  - ▶ **Confidencialidad:** Preservar la privacidad de la información, asegurando que solo usuarios autorizados tengan acceso ( Controles de Acceso y Cifrado).
  - ▶ **Integridad:** Garantizar que los datos sean precisos y no hayan sido alterados de manera no autorizada.
  - ▶ **Disponibilidad:** Asegurar que los datos estén disponibles para los usuarios autorizados cuando sea necesario (Redundancia y monitoreo constante).
  - ▶ **Autenticidad:** Evitar que una entidad niegue la autoría o recepción de una transacción (Firma Digital).

# Cifrado

- ▶ El cifrado de datos es la base principal de la seguridad de datos. Es un proceso que consiste en cifrar los datos para que solo los usuarios autorizados puedan acceder a ellos.
  - ▶ **Cifrado de Datos en Reposo:** Se cifran los datos almacenados físicamente en la base de datos. PostgreSQL admite la encriptación de archivos completos o columnas específicas.
  - ▶ **Cifrado en Transmisión:** Se cifran los datos en la transmisión entre el cliente y el servidor. PostgreSQL admite la encriptación de la conexión mediante el uso de SSL/TLS.
    - ▶ SSL (Secure Sockets Layer): Soporte nativo para cifrar las conexiones Cliente/Servidor. Se protegen los datos cuando son más vulnerables, es decir, durante la transmisión.
    - ▶ TLS (Transport Layer Security): Evolución de SSL, que aborda vulnerabilidades y debilidades encontradas en SSL.
- ▶ pg\_crypto es el módulo que soporta cifrado de contraseñas y de datos.

## 5.6 Monitorización y ajuste de Rendimiento

## Técnicas de Monitorización de Rendimiento

- ▶ Las bases de datos suelen mantener *logs* de consultas especialmente lentas. Es imprescindible monitorizar estos registros para entender qué está ocurriendo y que casos hay que mejorar.
  - ▶ Como es habitual, atacar primero el peor caso que ocurra con más frecuencia.
- ▶ Seguimiento continuo de parámetros esenciales de salud del servidor: *throughput*, tiempo de respuesta de consultas, almacenamiento en disco, utilización de espacio para indexación y operaciones específicas en consultas.
- ▶ Absolutamente **imprescindible** leer el manual del administrador, para configurar correctamente las variables operativas esenciales del servidor y clientes.

## Consultas Paralelizadas

- ▶ Algunas bases de datos como PostgreSQL pueden plantear planes de implementación de consultas que sacan partido de hardware multi-CPU.
- ▶ Esta característica se denomina consultas paralelizadas.
- ▶ Cuidado: no siempre es posible implementar consultas paralelizadas. Posibles motivos:
  - ▶ Limitaciones en la implementación actual.
  - ▶ Porque no se puede crear un plan para ejecutar la consulta de forma paralela más rápido que la consulta en serie.
- ▶ Muchas consultas que pueden sacar partido de esta característica pueden ir el doble de rápido, algunas hasta cuatro veces más deprisa.
- ▶ Las que más beneficio extraen son consultas que afectan a grandes cantidades de datos pero devuelven solo unas pocas filas.

## Consultas paralelizadas

- ▶ Cuando el optimizador determina que una consulta paralela es el plan de ejecución más rápido, crea un plan que incluye un nodo *Gather* o *Gather Merge*.

```
EXPLAIN SELECT * FROM pgbench_accounts WHERE filler LIKE '%x%';
QUERY PLAN
```

```
-----
Gather  (cost=1000.00..217018.43 rows=1 width=97)
Workers Planned: 2
->  Parallel Seq Scan on pgbench_accounts  (cost=0.00..216018.33 rows=1 width=97)
Filter: (filler ~~ '%x%'::text)
(4 rows)
```

- ▶ Control de número de *workers* en paralelo:
  - ▶ `max_parallel_workers_per_gather`, `max_worker_processes`, `max_parallel_workers`.

## 5.7 Sistemas Basados en Transacciones



# Transacciones y Propiedades ACID



- ▶ Las transacciones son unidades de trabajo que reúnen una serie de operaciones de lectura y escritura en la base de datos. Estas operaciones se ejecutan de manera coherente, es decir, todas o ninguna.
- ▶ Las transacciones aseguran la integridad y consistencia de los datos.
- ▶ Las transacciones en bases de datos se rigen por el modelo ACID, que garantiza las siguientes propiedades:
  - ▶ **Atomicidad:** Las operaciones se realizan como una unidad atómica.
  - ▶ **Consistencia:** La ejecución de una transacción lleva la base de datos de un estado consistente a otro consistente.
  - ▶ **Aislamiento (Isolation):** Cada transacción se ejecuta como si fuera la única, aislada de otras transacciones concurrentes.
  - ▶ **Durabilidad:** Una vez que una transacción se ha confirmado, sus efectos son permanentes y persisten incluso en caso de fallo del sistema.

# Implementación y manejo de Transacciones

- ▶ El manejo de transacciones implica el uso de sentencias SQL específicas, como **BEGIN TRANSACTION**, **COMMIT**, y **ROLLBACK**.
- ▶ Estas sentencias permiten iniciar, confirmar o deshacer una transacción, respectivamente. Además, las bases de datos proporcionan mecanismos para controlar el aislamiento y la durabilidad de las transacciones.

```
BEGIN;  
UPDATE stock SET quantity = quantity + 5 WHERE item_id = 2;  
COMMIT;
```

# Bibliografía I

-  [Connolly & Begg, 2015] Connolly, T., Begg, C.  
*Database Systems*.  
Pearson, 6th Global Edition. 2015.
-  [Petrov, 2019] Petrov, A.  
*Database Internals*.  
O'Reilly Media, 2019.
-  [Kraska et al., 2018] Kraska, T., Beutel, A., Chi, E.H., Dean, J., Polyzotis, N.  
*The Case for Learned Index Structures*.  
arXiv pre-print: URL<https://arxiv.org/abs/1712.01208>.

# Tema extra (Continuacion Tema 5) - Triggers y demás

David Moreno Lumbreras & Daniela Patricia Feversani

GSyC, EIF. URJC.

Laboratorio de Bases de Datos (BBDD)

Curso 2024-2025





(cc) 2020- David Moreno Lumbreras  
Algunos derechos reservados. Este trabajo se entrega bajo la licencia  
Creative Commons Reconocimiento - Compartirlgual  
(by-sa). Para obtener la licencia completa, véase  
<https://creativecommons.org/licenses/by-sa/3.0/es/>.

# Contenidos

E.1 Triggers

E.2 Procedimientos Almacenados

E.3 Transacciones (aun más)

## E.1 Triggers

# ¿Qué son los Triggers?

- ▶ Los **Triggers** son mecanismos de bases de datos que ejecutan acciones automáticamente cuando ocurre un evento específico (*INSERT*, *UPDATE*, *DELETE*) en una tabla.
- ▶ Tipos de triggers según el momento de ejecución:
  - ▶ **BEFORE**: Se ejecutan antes del evento.
  - ▶ **AFTER**: Se ejecutan después del evento.
- ▶ Se utilizan para:
  - ▶ Automatizar procesos.
  - ▶ Garantizar la integridad de los datos.
  - ▶ Registrar auditorías o aplicar reglas de negocio.



## Motivación para el Trigger: Auditoría en Clientes (BPSimple)

- ▶ **Objetivo:** Registrar los cambios realizados en los datos de los clientes para fines de auditoría.
- ▶ **Uso:** Controlar quién realiza los cambios y qué valores se modificaron.

# Auditoría de Cambios en Clientes (BPSimple)

Registrar actualizaciones en la tabla customer

```
CREATE TABLE customer_audit (  
    audit_id SERIAL PRIMARY KEY,  
    customer_id INT,  
    old_fname VARCHAR(32),  
    old_lname VARCHAR(32),  
    new_fname VARCHAR(32),  
    new_lname VARCHAR(32),  
    change_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP  
);  
  
CREATE OR REPLACE FUNCTION audit_customer_update()  
RETURNS TRIGGER AS $$  
BEGIN  
    INSERT INTO customer_audit (customer_id, old_fname, old_lname,  
                                new_fname, new_lname)  
    VALUES (OLD.customer_id, OLD.fname, OLD.lname, NEW.fname, NEW.lname);  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER after_update_customer  
AFTER UPDATE ON customer  
FOR EACH ROW  
EXECUTE FUNCTION audit_customer_update();
```

## Explicación del Trigger: Auditoría en Clientes

- ▶ Este trigger se ejecuta después de cada actualización en la tabla `customer`.
- ▶ Inserta un registro en la tabla `customer_audit` con los valores anteriores y nuevos de las columnas actualizadas.

## Motivación para el Trigger: Validación de Matrículas (Universidad)

- ▶ **Objetivo:** Evitar que un estudiante se matricule más de una vez en la misma asignatura.
- ▶ **Uso:** Garantizar la consistencia e integridad de los datos de matrículas.

# Validación de Matrículas (Universidad)

Evitar duplicados en la tabla Matriculas

```
CREATE OR REPLACE FUNCTION validar_matricula()
RETURNS TRIGGER AS $$
DECLARE
    matricula_existente INT;
BEGIN
    SELECT COUNT(*) INTO matricula_existente
    FROM Matriculas
    WHERE Matricula = NEW.Matricula AND IDAsignatura = NEW.IDAsignatura;

    IF matricula_existente > 0 THEN
        RAISE EXCEPTION 'Estudiante ya matriculado';
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER before_insert_matriculas
BEFORE INSERT ON Matriculas
FOR EACH ROW
EXECUTE FUNCTION validar_matricula();
```

## Explicación del Trigger: Validación de Matrículas

- ▶ Este trigger se ejecuta antes de insertar una nueva matrícula.
- ▶ Valida que no exista ya un registro para el estudiante y la asignatura.
- ▶ En caso de duplicado, lanza un error y evita la inserción.

## Motivación para el Trigger: Control de Inventario (DVD Rental)

- ▶ **Objetivo:** Actualizar automáticamente el inventario cuando se realiza un alquiler.
- ▶ **Uso:** Mantener consistencia en los datos del inventario sin necesidad de procesos manuales.

# Control de Inventario en Alquileres (DVDRentall)

Actualizar inventario en la tabla inventory

```
CREATE OR REPLACE FUNCTION actualizar_inventario()  
RETURNS TRIGGER AS $$  
BEGIN  
    UPDATE inventory  
    SET quantity = quantity - 1  
    WHERE inventory_id = NEW.inventory_id;  
  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER after_insert_rental  
AFTER INSERT ON rental  
FOR EACH ROW  
EXECUTE FUNCTION actualizar_inventario();
```



# Explicación del Trigger: Control de Inventario

- ▶ Este trigger se ejecuta después de insertar un nuevo alquiler.
- ▶ Decrementa automáticamente el inventario disponible para el artículo alquilado.
- ▶ Asegura que el inventario esté siempre actualizado.

## E.2 Procedimientos Almacenados

# Procedimientos Almacenados

- ▶ Los **procedimientos almacenados** (*Stored Procedures*) son bloques de código que se almacenan y se ejecutan directamente en la base de datos.
- ▶ Se usan para automatizar procesos repetitivos, reglas de negocio y operaciones complejas.
- ▶ Ventajas principales:
  - ▶ Mejora del rendimiento al reducir el tráfico entre cliente y servidor.
  - ▶ Reutilización del código.
  - ▶ Facilita la gestión y el mantenimiento de reglas de negocio.
- ▶ Soportan parámetros de entrada y salida.

## Motivación para el Procedimiento: Total de Ventas por Cliente (BPSimple)

- ▶ **Objetivo:** Calcular el total de ventas realizadas por un cliente.
- ▶ **Uso:** Generar reportes rápidos sobre el comportamiento de compra de los clientes.

## Total de Ventas por Cliente (BPSimple)

```
CREATE OR REPLACE FUNCTION obtener_total_ventas(cliente_id INT)
RETURNS NUMERIC AS $$
DECLARE
    total NUMERIC;
BEGIN
    SELECT SUM(total_amount) INTO total
    FROM invoice
    WHERE customer_id = cliente_id;

    RETURN total;
END;
$$ LANGUAGE plpgsql;
```

### Ejecución:

```
SELECT obtener_total_ventas(1);
```

## Explicación del Procedimiento: Total de Ventas por Cliente

- ▶ AS \$\$ ... \$\$ Es un delimitador utilizado para encapsular el bloque de código.
- ▶ LANGUAGE plpgsql; Especifica el lenguaje de programación que se usará para definir el procedimiento o función, en este caso es el lenguaje de procedimientos de PostgreSQL basado en SQL.
- ▶ Este procedimiento recibe el ID de un cliente como parámetro de entrada.
- ▶ Calcula la suma del campo `total_amount` de la tabla `invoice` para ese cliente.
- ▶ Devuelve el total como un valor de retorno.

## Motivación para el Procedimiento: Promedio de Notas por Estudiante (Universidad)

- ▶ **Objetivo:** Calcular el promedio de las calificaciones obtenidas por un estudiante en todas sus asignaturas.
- ▶ **Uso:** Evaluar el desempeño académico de los estudiantes.

# Promedio de Notas por Estudiante (Universidad)

```
CREATE OR REPLACE FUNCTION calcular_promedio_notas(matricula INT)
RETURNS NUMERIC AS $$
DECLARE
    promedio NUMERIC;
BEGIN
    SELECT AVG(Calificacion) INTO promedio
    FROM Matriculas
    WHERE Matricula = matricula;

    RETURN promedio;
END;
$$ LANGUAGE plpgsql;
```

## Ejecución:

```
SELECT calcular_promedio_notas(1001);
```



## Explicación del Procedimiento: Promedio de Notas por Estudiante

- ▶ Este procedimiento recibe el ID de un estudiante (Matricula) como parámetro de entrada.
- ▶ Calcula el promedio de las calificaciones almacenadas en la tabla Matriculas para ese estudiante.
- ▶ Devuelve el promedio como un valor de retorno.

## Motivación para el Procedimiento: Actualizar Inventario (DVD Rental)

- ▶ **Objetivo:** Disminuir la cantidad disponible de un artículo tras realizar un alquiler.
- ▶ **Uso:** Mantener actualizado el inventario de manera eficiente.

# Actualizar Inventario (DVD Rental)

```
CREATE OR REPLACE PROCEDURE actualizar_inventario(inventario_id INT)
LANGUAGE plpgsql
AS $$
BEGIN
    UPDATE inventory
    SET quantity = quantity - 1
    WHERE inventory_id = inventario_id;
END;
$$;
```

## Ejecución:

```
CALL actualizar_inventario(20);
```

## Explicación del Procedimiento: Actualizar Inventario

- ▶ Este procedimiento recibe el ID del inventario como parámetro de entrada.
- ▶ Disminuye en 1 la cantidad disponible (`quantity`) para el artículo correspondiente en la tabla `inventory`.
- ▶ Ayuda a mantener el inventario sincronizado tras un alquiler.

## E.3 Transacciones (aun más)

## Conceptos clave

- ▶ Una **transacción** es una unidad de trabajo que agrupa una o más operaciones de base de datos, tratándolas como una sola operación lógica.
- ▶ Propiedades fundamentales: **ACID**.
  - ▶ **Atomicidad**: La transacción es indivisible; o se ejecuta por completo o no se ejecuta.
  - ▶ **Consistencia**: La base de datos pasa de un estado válido a otro estado válido.
  - ▶ **Aislamiento**: Las transacciones concurrentes no interfieren entre sí.
  - ▶ **Durabilidad**: Los cambios realizados por una transacción confirmada son permanentes.
- ▶ Las transacciones son gestionadas con BEGIN, COMMIT y ROLLBACK.

# Manejo de Transacciones con BEGIN, COMMIT y ROLLBACK

- ▶ **BEGIN:** Indica el inicio de una transacción.
- ▶ **COMMIT:** Confirma los cambios realizados durante la transacción, haciéndolos permanentes.
- ▶ **ROLLBACK:** Revierte todos los cambios realizados desde el inicio de la transacción.
- ▶ **Casos comunes de errores:**
  - ▶ Fallo de integridad referencial (clave foránea).
  - ▶ Violación de restricciones UNIQUE o NOT NULL.
  - ▶ Errores en los cálculos o lógica de negocio.
- ▶ Si ocurre un error en cualquier paso, ROLLBACK asegura que la base de datos vuelva a su estado original.

## Motivación para el Uso de Transacciones: Gestión de Clientes (BPSimple)

- ▶ **Objetivo:** Asegurar la integridad de los datos al actualizar información de un cliente.
- ▶ **Uso:** Evitar inconsistencias como actualizar parcialmente los datos de un cliente.



## Gestión de Clientes (BPSimple)

```
BEGIN;  
  
-- Actualizar nombre y direccion del cliente  
UPDATE customer  
SET fname = 'John',  
    address = '123 New Street'  
WHERE customer_id = 1;  
  
-- Confirmar la transaccion  
COMMIT;
```

**Si ocurre un error:**

```
ROLLBACK;
```

## Explicación de la Transacción: Gestión de Clientes

- ▶ La transacción asegura que todos los cambios en los datos del cliente se realicen juntos.
- ▶ Si ocurre un error, todos los cambios se revierten.

# Motivación para el Uso de Transacciones: Gestión de Matrículas (Universidad)

- ▶ **Objetivo:** Asegurar que los datos de matrícula y las actualizaciones relacionadas sean consistentes.
- ▶ **Uso:** Garantizar que las asignaturas se registren correctamente y se actualicen los cupos.

# Gestión de Matrículas (Universidad)

```
BEGIN;  
  
-- Insertar matricula del estudiante  
INSERT INTO Matriculas (Matricula, IDAsignatura, Fecha)  
VALUES (1001, 101, CURRENT_DATE);  
  
-- Reducir cupo disponible de la asignatura  
UPDATE Asignaturas  
SET CuposDisponibles = CuposDisponibles - 1  
WHERE IDAsignatura = 101;  
  
-- Confirmar la transaccion  
COMMIT;
```

**Si ocurre un error:**

```
ROLLBACK;
```

# Explicación de la Transacción: Gestión de Matrículas

- ▶ La transacción asegura que la matrícula y la reducción del cupo se ejecuten juntas.
- ▶ Si ocurre un error al actualizar el cupo, la matrícula también se revierte.

## Motivación para el Uso de Transacciones: Gestión de Inventario (DVD Rental)

- ▶ **Objetivo:** Asegurar que el registro de un alquiler y la actualización del inventario sean consistentes.
- ▶ **Uso:** Evitar errores como registrar un alquiler sin descontar el inventario.

# Gestión de Inventario en Alquileres (DVD Rental)

```
BEGIN;  
  
-- Insertar nuevo alquiler  
INSERT INTO rental (rental_date, inventory_id, customer_id, return_date)  
VALUES (CURRENT_DATE, 10, 1, NULL);  
  
-- Reducir inventario disponible  
UPDATE inventory  
SET quantity = quantity - 1  
WHERE inventory_id = 10;  
  
-- Confirmar la transaccion  
COMMIT;
```

**Si ocurre un error:**

```
ROLLBACK;
```

## Explicación de la Transacción: Gestión de Inventario

- ▶ La transacción asegura que el registro del alquiler y la reducción del inventario se realicen juntos.
- ▶ Si ocurre un error en cualquiera de los pasos, todos los cambios se revierten.



# Bibliografía I



[Connolly & Begg, 2015] Connolly, T., Begg, C.  
*Database Systems*.  
Pearson, 6th Global Edition. 2015.



[Petrov, 2019] Petrov, A.  
*Database Internals*.  
O'Reilly Media, 2019.

## Tema 6 - Formatos de representación de datos

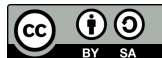
David Moreno Lumbreras & Daniela Patricia Feversani

GSyC, EIF. URJC.

Laboratorio de Bases de Datos (BBDD)

Curso 2024-2025





(cc) 2020- David Moreno Lumbreras  
Algunos derechos reservados. Este trabajo se entrega bajo la licencia  
Creative Commons Reconocimiento - Compartirlgual  
(by-sa). Para obtener la licencia completa, véase  
<https://creativecommons.org/licenses/by-sa/3.0/es/>.

# Contenidos

6.1 Formatos de representación

6.2 Datos CSV

6.3 Gestión de datos JSON

6.4 Gestión de datos XML

6.5 Gestión de datos geográficos

6.6 Otros formatos de representación de datos

## 6.1 Formatos de representación

# Introducción a los formatos de datos

- ▶ **¿Qué son los formatos de datos?:** Los formatos de datos son las convenciones utilizadas para representar datos. Estos formatos pueden variar en términos de estructura, esquema y flexibilidad.
- ▶ **Importancia de los formatos de datos:** Los formatos de datos son fundamentales en la gestión de bases de datos ya que determinan cómo se almacenan, se organizan y se accede a los datos. Elegir el formato de datos adecuado puede tener un impacto significativo en el rendimiento de una base de datos.
- ▶ **Tipos de formatos de datos:** Los formatos de datos se pueden clasificar en tres categorías principales: datos estructurados, datos semi-estructurados y datos no estructurados. Cada uno tiene sus propias características y usos.

# Tipos de datos

- ▶ **Datos estructurados:** Están sujetos a un **esquema fijo** de representación, conocido de antemano. Los datos se debe ajustar obligatoriamente a ese esquema.
  - ▶ Ejemplos: base de datos relacional, CSV ('Comma-Separated Values')/TSV('Tab-Separated Values').
  - ▶ A veces aparece el término datos tabulados como sinónimo.
- ▶ **Datos semi-estructurados:** Están sujetos a un **esquema flexible** de representación. Se definen los campos que almacenan la información y el tipo de dato de cada campo, así como atributos que ofrecen metadatos adicionales. Puede que falte algún campo y/o atributo (aunque se puede forzar a que se cumpla el esquema estrictamente).
  - ▶ Ejemplos: XML (Extensible Markup Language), JSON/JSONB (JavaScript Object Notation).
- ▶ **Datos no estructurados:** Se pueden interpretar siguiendo unas conjunto de reglas, pero la estructura y organización de su contenido es libre y no se conoce de antemano.
  - ▶ Ejemplo: texto en lenguaje natural (Documentos, correos electrónicos).

# Datos en RDBMS

- ▶ Las bases de datos relacionales surgieron, originalmente, para almacenar datos estructurados (esquema fijo). Una vez acordados los campos de cada tabla y el tipo de dato de cada uno, no se puede cambiar.
- ▶ Si se cambia el esquema de una tabla, entonces hay que reconstruirla por completo.
- ▶ Posteriormente, muchas bases de datos relacionales han evolucionado para poder almacenar datos semi-estructurados o incluso no estructurados (con ciertas limitaciones). Por ejemplo:
  - ▶ PostgreSQL: <https://www.postgresql.org/docs/current/datatype.html>. Incluye soporte para datos semi-estructurados, como JSON o XML.
  - ▶ Oracle: Incluye soporte para datos semi-estructurados, como XML o JSON. Además, en sus versiones más recientes, ofrece extensiones para realizar búsquedas de texto en grandes bloques de caracteres (CLOB), es decir, datos no estructurados.



# Formatos de representación de datos

Además de los tipos de datos especiales incluidos en PostgreSQL (geométricos, direcciones IP, etc.), existen otros estándares de representación de datos:

- ▶ CSV / TSV.
- ▶ JSON / JSONB (JSON binario).
- ▶ YAML.
- ▶ XML.
- ▶ Datos GIS (Geographical Information Systems).
- ▶ Formatos en sistemas distribuidos: Apache Parquet, Apache Avro, Apache Arrow ...

## 6.2 Datos CSV

# Comma-separated values (CSV)

Formato clásico de representación de datos estructurados.

- ▶ Es un formato altamente flexible que permite ajustarse a las necesidades particulares de los usuarios o las aplicaciones.
- ▶ Se pueden definir múltiples *dialectos*: carácter separador de items en una fila, carácter delimitador de *strings*, carácter separador de decimales, formas de escapar caracteres especiales, etc. No es, ni mucho menos, universal.
- ▶ Todavía hoy día supone, en muchos casos, la forma más rápida de volcar o cargar datos en un sistema de gestión de datos estructurados (e.g. una base de datos relacional).
- ▶ Es un formato estandarizado ([RFC 4180](#)), aunque en la práctica no siempre se emplea.
- ▶ Cada **registro** (fila) contiene el mismo número de campos, separador por delimitadores. Deben aparecer siempre en el mismo orden. Si falta el valor para algún campo se deja el hueco entre dos delimitadores (*datos faltantes*).

## 6.3 Gestión de datos JSON

# JavaScript Object Notation (JSON)

Formato estándar de representación de datos semi-estructurados en la web.

- ▶ No tienen un esquema definido, sino que se organizan mediante etiquetas que permiten agruparlos y crear jerarquías.
- ▶ Es un formato de intercambio de datos ligero y fácilmente legible por humanos.
- ▶ Su sintaxis está inspirada en la notación de objetos de JavaScript, pero es un formato independiente del lenguaje.

# JavaScript Object Notation (JSON)

- ▶ Estándar abierto: <https://json.org/>, ECMA-404.
- ▶ Versión actual: RFC 8259.
- ▶ Se representan los datos como pares clave-valor.
  - ▶ Tipos de datos: numérico, *string*, booleano, array, objeto.
  - ▶ Un objeto se delimita por {}, un array se delimita con [].
  - ▶ Separador de nombres con : y separador de valores con ,.

# JSON: ejemplo

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

# JSON: JSON Schema

- ▶ **JSON Schema:** Herramienta empleada para validar estructuras de datos JSON.
- ▶ Consiste en un vocabulario que se puede emplear para establecer consistencia, validez e interoperabilidad a los datos JSON.
- ▶ Establece un lenguaje común para intercambiar datos, definiendo reglas de validación precisas para las estructuras de datos.
- ▶ Un esquema JSON define las reglas que deben seguir los datos JSON. Se emplean validadores de esquemas para comprobar que los datos cumplen con el esquema.
- ▶ JSON Schema está estandarizado (Versión 2020-12). [JSON Schema](#).



# JSON: JSON Schema

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Persona",
  "type": "object",
  "properties": {
    "nombre": {
      "type": "string"
    },
    "edad": {
      "type": "integer",
      "minimum": 0
    },
    "ciudad": {
      "type": "string"
    }
  },
  "required": ["nombre", "edad"]
}
```

```
{
  "nombre": "Juan",
  "edad": 30,
  "ciudad": "Madrid"
}

{
  "nombre": "Juan",
  "ciudad": "Madrid"
}
```

# JSON en PostgreSQL

- ▶ Los formatos de datos JSON y JSONB están soportados desde la versión 9.4.
- ▶ PostgreSQL tiene dos tipos de datos para almacenar información JSON: **JSON** y **JSONB**.
  - ▶ **JSON**: Almacena una copia exacta del texto de entrada JSON. Incluye espacios en blanco o el orden de las claves, pero no permite la indexación.
  - ▶ **JSONB**: Almacena datos JSON en un formato binario descompuesto. Esto significa que se almacenan de una manera que permite una búsqueda y manipulación más eficientes.
    - ▶ No conserva espacios en blanco, ni orden de las claves, ni claves duplicadas.
    - ▶ Permite la indexación.
    - ▶ Se recomienda usar JSONB debido a su eficiencia.
- ▶ JSON Types.
- ▶ JSON functions and operators.

## 6.4 Gestión de datos XML

# eXtensible Markup Language (XML)

- ▶ Metalenguaje (lenguaje para describir otros lenguajes), que permite a los diseñadores crear sus propias etiquetas personalizadas para proporcionar funcionalidad no disponible en HTML.
- ▶ XML fue diseñado para almacenar y transportar datos.
- ▶ Es autodescriptivo, es decir, las etiquetas definidas son las encargadas de proporcionar un contexto a los datos.
- ▶ Es un estándar abierto, simple, extensible, reutilizable, que separa contenido de la presentación del mismo.
- ▶ Almacena datos en un formato de texto plano, lo que proporciona una forma independiente de Software y Hardware para almacenar, transportar y compartir datos.
- ▶ Soporte en bases de datos relacionales: estándar SQL/XML (Desde el estándar 2003 y 2006).

# Estructura documento XML

- ▶ Los datos XML presentan una estructura jerárquica formada por un conjunto de elementos correctamente anidados que no se solapan entre ellos.
  - ▶ **Elemento Raíz o Root:** Es un elemento único en el XML que se encarga de contener a todos los demás.
  - ▶ **Prólogo o Declaración:** sección inicial, opcional. Indica la versión de XML, codificación y si se debe chequear contra una DTD. También puede indicar enlaces a *stylesheet* y documento DTD.
  - ▶ **Etiquetas:** Construcción de marcado que comienza con < y termina con > y que, distingue entre mayúsculas y minúsculas. Cada elemento de un documento XML debe estar delimitado por una etiqueta de inicio y una etiqueta de fin.
  - ▶ **Entidades:** XML ofrece métodos (entidades) para referir a algunos caracteres especiales reservados.
    - ▶ < → &lt;
    - ▶ > → &gt;
    - ▶ & → &amp;
    - ▶ ' → &apos;
    - ▶ " → &quot;

# XML: ejemplo

```
<?xml version= "1.1" encoding= "UTF-8" standalone= "no"?>
<?xml:stylesheet type = "text/xsl" href = "staff_list.xsl"?>
<!DOCTYPE STAFFLIST SYSTEM "staff_list.dtd">
<STAFFLIST>
  <STAFF branchNo = "B005">
    <STAFFNO>SL21</STAFFNO>
    <NAME>
      <FNAME>John</FNAME><LNAME>White</LNAME>
    </NAME>
    <POSITION>Manager</POSITION>
    <DOB>1945-10-01</DOB>
    <SALARY>30000</SALARY>
  </STAFF>
  <STAFF branchNo = "B003">
    <STAFFNO>SG37</STAFFNO>
    ...
  </STAFF>
</STAFFLIST>
```

# XML: Definición de Tipo de Documento

- ▶ Para ser válido, un documento XML necesita cumplir ciertas reglas de semántica que se definen en un esquema XML o en una Definición de Tipo de Documento (*DTD*).
- ▶ El documento DTD indica el formato y número de ocurrencias que pueden aparecer de un elemento, siendo ocurrencias el número de veces que un elemento específico puede aparacer en el XML:
  - ▶ Un \* indica cero o más ocurrencias de un elemento.
  - ▶ Un + indica una o más ocurrencias de un elemento.
  - ▶ Un ? indica cero o *exactamente* una ocurrencia.
  - ▶ Cualquier elemento sin cualificador debe ocurrir exactamente una vez.
  - ▶ #PCDATA indica datos en formato caracter parseable.
  - ▶ CDATA indica que se pase el texto delimitado directamente a la aplicación, sin interpretación.

## DTD: ejemplo

*<!-- DTD para el documento XML del ejemplo previo -->*

*<!ELEMENT STAFFLIST (STAFF)\*>*

*<!ELEMENT STAFF (NAME, POSITION, DOB?, SALARY)>*

*<!ELEMENT NAME (FNAME, LNAME)>*

*<!ELEMENT FNAME (#PCDATA)>*

*<!ELEMENT LNAME (#PCDATA)>*

*<!ELEMENT POSITION (#PCDATA)>*

*<!ELEMENT DOB (#PCDATA)>*

*<!ELEMENT SALARY (#PCDATA)>*

*<!ATTLIST STAFF branchNo CDATA #IMPLIED>*



# XML: otras tecnologías

- ▶ Existen un conjunto de tecnologías y estándares que permiten trabajar con documentos XML.
  - ▶ Interfaces (APIs): DOM (basada en árbol, vista orientada a objetos) y SAX (basada en eventos, acceso serializado). Permiten a los programas interactuar con documentos XML.
  - ▶ *Namespaces*: Nombres de elementos y relaciones cualificados, para evitar colisiones en elementos con mismo nombre y definiciones distintas.
  - ▶ XSLT: Mecanismos para transformación de XML en otros formatos, como HTML o SQL.
  - ▶ XPath: Lenguaje de consulta declarativo para recuperar elementos de un documento XML.
  - ▶ XPointer: Proporciona acceso a los valores de atributos o el contenido de elementos en cualquier lugar de un documento XML.
  - ▶ XML Schema: Permite definir la estructura de un documento XML de forma más robusta que con DTD. El esquema está escrito en XML.

# Soporte XML en PostgreSQL

Resumen: [PostgreSQL vs SQL/XML Standards](#).

- ▶ PostgreSQL incluye el [tipo de dato XML](#).
- ▶ También incluye una [lista de funciones para XML](#), conformes con el estándar SQL/XML.
- ▶ Igualmente, incluye funciones para mapear una tabla, esquema o catálogo de la base de datos a un documento XML y un XML Schema.
- ▶ Incluye, también, varias características adicionales y funciones, más allá de las definidas en el estándar.

```
SELECT xpath('/libro/titulo/text()', contenido_xml)
FROM biblioteca
WHERE id = 1;
```

## 6.5 Gestión de datos geográficos

# Sistema de Información Geográfica (GIS)

- ▶ Un Sistema de Información Geográfica (GIS en inglés), debe implementar una serie de características y servicios para manipulación de datos geográficos:
  - ▶ Lectura, edición, almacenamiento y gestión (en general), de datos espaciales.
  - ▶ Análisis de esos datos, incluyendo consultas sencillas o elaboración de modelos complejos, tanto sobre la componente espacial de los datos (localización de cada valor o elemento) como sobre la componente temática (valor o elemento en sí).
  - ▶ Generación de resultados: informes, gráficos, mapas, etc.
- ▶ En su forma básica, SQL no recoge las geometrías que forman la parte espacial de una entidad. Para estandarizar, Simple Features for SQL (SFS) define tipos estandarizados para geometrías, basados en [OpenGIS Geometry Model](#).

Libro en español: “[Sistemas de Información Geográfica](#)” - V. Olaya. (CC-BY, 2020).

# PostGIS

- ▶ PostGIS es una extensión para PostgreSQL que añade soporte para objetos geográficos, permitiendo consultas de localización en formato SQL.
- ▶ <https://postgis.net/>.
- ▶ También se incluyen muchas otras características que, según la página del proyecto, están raramente presentes en otras bases de datos espaciales (como Oracle Locator/Spatial y SQL Server).
  - ▶ Listado completo de características.
- ▶ Manual completo (v3.1).

# PostGIS

- ▶ Incluye el tipo Geography, con soporte nativo para características espaciales representadas por coordenadas geográficas (lat/lon). Son coordenadas esféricas expresadas en unidades angulares.
- ▶ Posteriormente, se pueden efectuar [consultas SQL](#) para recuperar los datos espaciales.
- ▶ Igualmente, también se pueden volcar los datos a un archivo *shape*, mediante la herramienta `pgsql2shp`.
- ▶ Algunos índices de PostgreSQL están específicamente pensados para trabajar con datos espaciales:
  - ▶ GiST.
  - ▶ BRIN.
  - ▶ SP-GiST.

## 6.6 Otros formatos de representación de datos

# Apache Parquet

- ▶ <https://parquet.apache.org/>.
- ▶ Formato de almacenamiento de datos *columnar*, que incluye una serie de optimizaciones especialmente diseñadas para cargas de trabajo de análisis de datos.
- ▶ Comprime los datos de columnas, ahorrando espacio de almacenamiento y permitiendo lectura de columnas individuales en lugar de archivos completos.
- ▶ La lectura de datos en formato Parquet es, casi siempre, mucho más eficiente que en formato CSV o JSON.
- ▶ También soporta tipos de datos complejos: array, *map* o *struct*



# Apache Avro

- ▶ <https://avro.apache.org/>.
- ▶ Formato de serialización de datos, frecuentemente ligado a Apache Hadoop.
- ▶ Incluye un framework para RPCs y formato de serialización orientado a *filas*.
- ▶ El formato de serialización permite tanto almacenamiento persistente de datos como envío a través del cable entre nodos de un cluster.
- ▶ El esquema de definición de la estructura de datos se puede realizar en JSON.
- ▶ Es similar a otras alternativas como Thrift y Protocols Buffers.

# Apache Arrow

- ▶ <https://arrow.apache.org/>.
- ▶ Formato de representación de datos columnar en memoria, independiente de lenguaje.
- ▶ Soporta datos planos o jerárquicos y está optimizado para CPUs y GPUs modernas.
- ▶ Soporta *zero-copy reads*: el procesador no participa en la copia de datos de un área de memoria a otra.
- ▶ Librerías disponibles en múltiples lenguajes.
- ▶ Arrow vs Parquet.

# Bibliografía I



[Connolly & Begg, 2015] Connolly, T., Begg, C.  
*Database Systems*.  
Pearson, 6th Global Edition. 2015.



[Petrov, 2019] Petrov, A.  
*Database Internals*.  
O'Reilly Media, 2019.