

# Apunte Base de Datos

Victoria Fiora

20 de julio de 2025

## 1. Generalidades de la Materia

### Esquema General

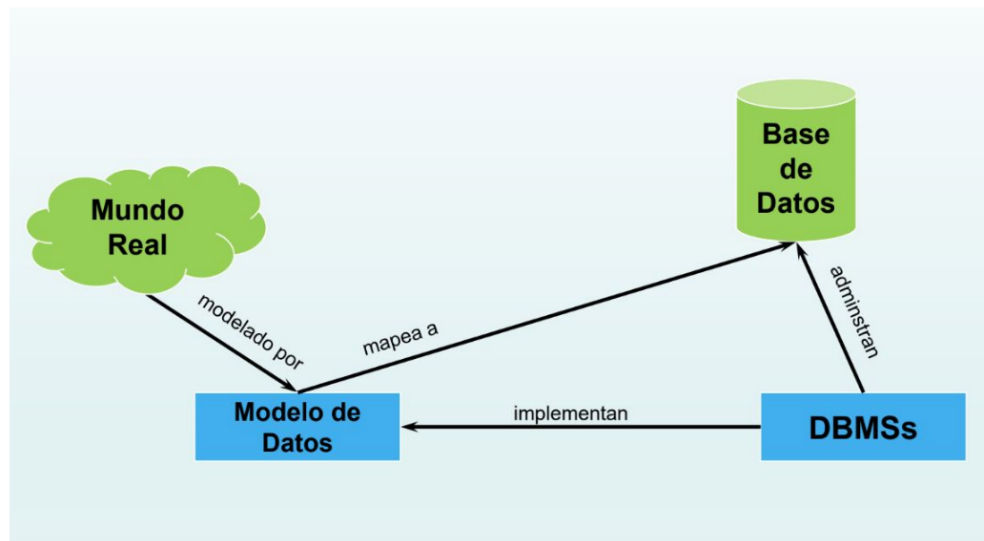


Figura 1: Relación entre el mundo real, el modelo de datos, los DBMS y la base de datos

- El **Mundo Real** es modelado a través de un **Modelo de Datos**, que actúa como una abstracción formal.
- Este modelo se **mapea** a una **Base de Datos** concreta que contiene los datos instanciados.
- Los **DBMSs (Database Management Systems)** implementan el modelo de datos y administran la base de datos. Es la herramienta que utiliza cada aplicación para manejar grandes cantidades de datos de manera eficiente

Otros componentes de la DBMS incluyen:

- **Recovery Manager:** Encargado de restaurar la base de datos a un estado consistente en caso de haber una falla. Para hacer eso hace uso del log, un archivo que lleva un registro de las acciones efectuadas a la DB.
- **System catalog:** es el lugar donde un DBMS guarda los metadatos del esquema. Los metadatos pueden incluir:
  - Información sobre las tablas y columnas
  - Views
  - índices
  - Usuarios y grupos de usuario.
  - Triggers
  - Funciones de agregación definidas por el usuario.
- **Optimizador de consultas:** Encargado de armar un plan de ejecución eficiente en base a una consulta, basándose en la información del system catalog.

## Distinción Conceptual

- **Base de datos:** Es un conjunto de datos relacionados con un significado inherente.
- **Datos:** hechos o registros con significado implícito.
- **Metadatos:** datos que describen otros datos.
- **Información:** datos procesados y organizados.
- **Conocimiento:** aplicación de la información para toma de decisiones.
- **Sabiduría:** uso experto del conocimiento.

## Proceso de Creación de una Base de Datos

El desarrollo de una base de datos sigue una serie de etapas estructuradas:

1. **Análisis de Requerimientos:** Identificación de las necesidades del sistema, el tipo de datos a almacenar y las consultas que deberán realizarse.
2. **Modelo Entidad-Relación (MER):** Representa de forma conceptual los objetos de interés y sus relaciones, como una abstracción del dominio del problema.
3. **Modelo Relacional (MR):** Conversión del MER a esquemas relacionales.
4. **Normalización:** Proceso que asegura que las relaciones no presenten redundancia ni anomalías de actualización, aplicando formas normales.
5. **Diseño Físico:** Definición de tipos de datos, índices, organización de almacenamiento y optimizaciones según el SGBD.

## Arquitectura e Independencia

En una base de datos se reconocen tres niveles:

- **Nivel Interno (o físico):** Describe el almacenamiento físico de las estructuras de la base de datos.
- **Nivel Conceptual (o lógico):** Contiene el esquema conceptual que describe la estructura de la base de datos sin enfocarse en lo físico, sino en entidades, tipos de datos, operaciones de usuario y restricciones.
- **Nivel Externo (o de usuario):** Describe las vistas o esquemas de usuario que muestran la parte de la base de datos que le interesa a un grupo en particular.

Entre estos niveles existen las siguientes formas de independencia:

- **Independencia lógica:** Es la capacidad de cambiar el esquema conceptual sin modificar los esquemas externos. Esto incluye expandir o reducir la base de datos o cambiar restricciones. No suele ser fácil de lograr. Implica que la capa lógica o conceptual sea independiente de la capa externa, permitiendo que el usuario no necesite conocer la arquitectura lógica de los datos para utilizar la base de datos.
- **Independencia física:** Es la capacidad de modificar el esquema interno sin alterar el esquema conceptual (y, por ende, los externos). Esto puede incluir la reorganización de archivos o la incorporación de índices para mejorar las consultas. De este modo, es posible comprender y modificar la base de datos en términos conceptuales sin necesidad de entender su almacenamiento físico.

## A.C.I.D.: Propiedades de las Bases de Datos Transaccionales

- **Atomicidad:** una operación compuesta se ejecuta completamente o no se ejecuta nada.
- **Consistencia:** las lecturas retornan siempre el valor más reciente del ítem leído.
- **Aislamiento:** las operaciones concurrentes no interfieren entre sí.
- **Durabilidad:** los cambios confirmados persisten ante fallos del sistema.

## Componentes del Lenguaje SQL

- **DDL (Data Definition Language):** Es el lenguaje utilizado para **definir la estructura de la base de datos** o los esquemas. Comandos: CREATE, ALTER, DROP.
- **DML (Data Manipulation Language):** Es el lenguaje para **consultar y modificar datos** dentro de las tablas. Comandos: SELECT, INSERT, UPDATE, DELETE, TRUNCATE.
- **DCL (Data Control Language):** Es el lenguaje usado para **gestionar permisos y accesos** a los objetos de la base de datos. Comandos: GRANT, REVOKE.
- **TCL (Transaction Control Language):** Es el lenguaje para **controlar transacciones** y confirmar o deshacer cambios. Comandos: BEGIN TRANSACTION, COMMIT, ROLLBACK.

## 2. Modelo Entidad-Relación

### Modelado Conceptual

- Es una conceptualización formal del mundo real (dominio específico) que modela sus objetos, características y relaciones.
- Usa **entidades** (cosas del mundo real), **atributos** (propiedades) y **relaciones** (asociaciones entre entidades).
- Proporciona una representación abstracta que no depende de implementaciones físicas.
- La **dependencia funcional** es una restricción entre atributos:  $X \rightarrow Y$  significa que si dos tuplas coinciden en  $X$ , deben coincidir en  $Y$ .
- Esta restricción es propiedad de los datos, no del diseño deseado.

### Modelo Entidad-Relación (MER)

Es una herramienta que permite realizar una abstracción o modelo de alguna situación de interés del mundo real. Se realiza a través de la técnica de los DERs (Diagramas de Entidad-Relación) conformados por los siguientes elementos:

- **Entidad:** conjunto de objetos distinguibles. Puede ser fuerte o débil.
- **Atributos:** Son las características descriptivas de las entidades relevantes al problema. Pueden ser de varios tipos:
  - **Simples:** No se dividen en partes más pequeñas (por ejemplo, **nombre** o **edad**).
  - **Compuestos:** Pueden descomponerse en subatributos (por ejemplo, **dirección** en **calle**, **número** y **ciudad**).
  - **Multivaluados:** Pueden tomar varios valores para una misma entidad (por ejemplo, varios números de teléfono de una persona).
  - **Derivados:** Su valor se calcula a partir de otros atributos (por ejemplo, **edad** a partir de **fecha de nacimiento**).
- **Superclave(SK):** cualquier conjunto de atributos que determina todos los atributos de una entidad. Puede contener atributos redundantes.
- **Clave candidata(CK):** conjunto mínimo de atributos que determina todos los atributos de la entidad. Formalmente:
  - $X \rightarrow R$  ( $X$  determina todos los atributos de  $R$ )
  - No existe  $Y \subset X$  tal que  $Y \rightarrow R$  (minimalidad)
- **Clave primaria(PK):** Una CK elegida según un criterio para identificar a la entidad.

## Restricciones y Cardinalidad

- **Cardinalidad:** especifica el número de entidades que pueden participar en una relación.
- **Restricción de participación:** **Total** significa que cada instancia de la entidad participa obligatoriamente en la relación. **Parcial** significa que no todas las instancias participan.



Figura 2: La entidad 1 participa de manera **total** en la interrelación. La entidad 2 participa de forma **parcial**.

## Entidad Débil y Clave Parcial

- Una entidad débil no tiene clave primaria propia.
- Se identifica mediante una **clave parcial** y la clave de la entidad fuerte.
- La relación que la vincula debe ser de participación total.

## Agregación

- La agregación permite modelar el hecho de que una **relación completa** pueda participar en otra relación.
- En una **relación ternaria** los tres elementos deben participar **simultáneamente**; en cambio, la agregación permite **flexibilizar** esa condición, tratando la relación compuesta como una entidad y permitiendo su participación parcial en otras relaciones.

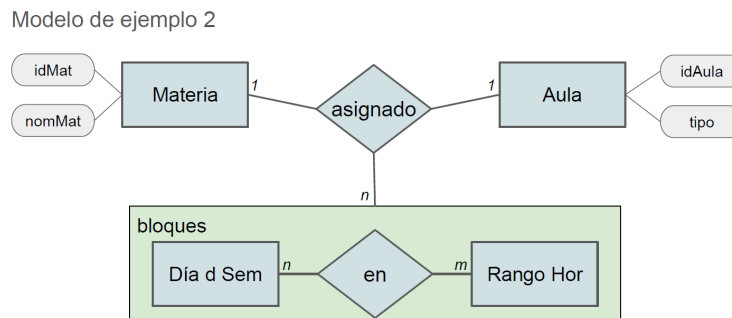


Figura 3: Ejemplo de agregación: una relación participando en otra relación

## Jerarquías y Generalización

- Una jerarquía modela la relación entre una entidad general (superclase) y entidades más específicas (subclases).
- Puede ser **total** (toda entidad general pertenece a una subclase) o **parcial** (no todas lo hacen).
- Puede ser **disjunta** (una instancia sólo pertenece a una subclase) o **solapada** (puede pertenecer a varias).
- Se usa para representar herencia y especialización en el modelo E-R.

## Trampas de Conexión

**Trampa del abanico** (Fan Traps): El camino entre ciertas entidades es ambiguo. Sucede generalmente cuando salen dos o mas interrelaciones 1:N en abanico desde la misma entidad.

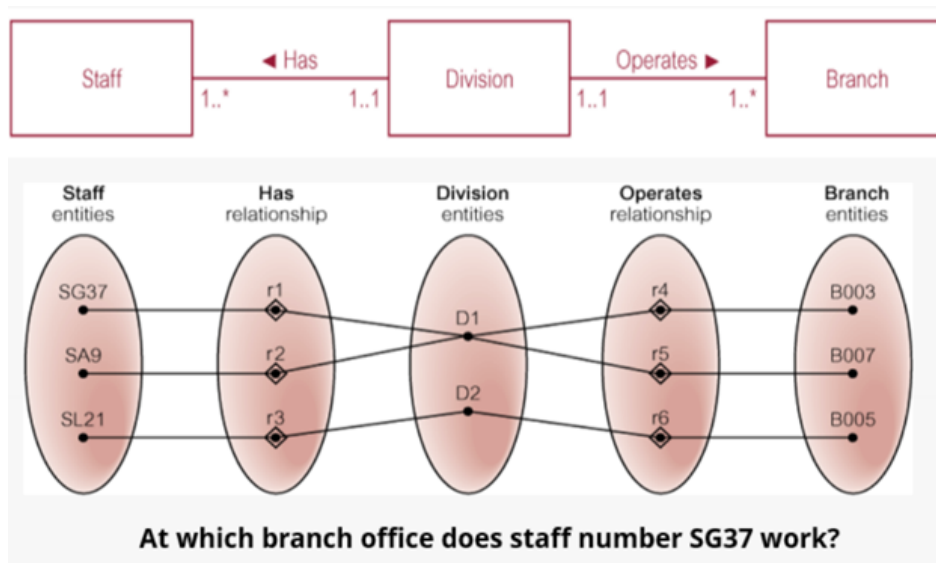


Figura 4: Trampa Abanico Problema

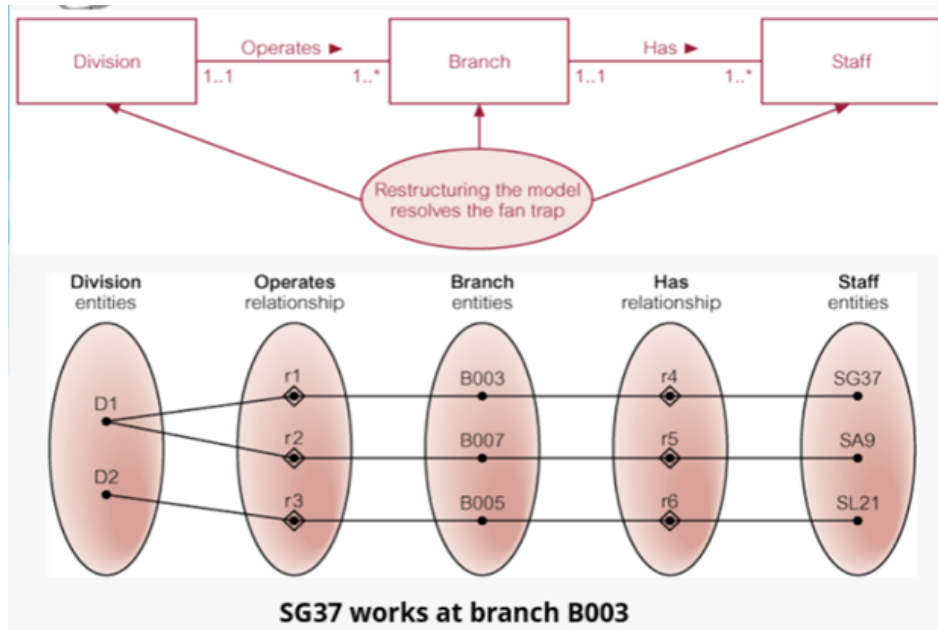


Figura 5: Trampa Abanico Solución

**Trampa del sumidero** (Chasm Traps): La interrelación existente entre dos entidades del modelo no tiene camino. Suele aparecer cuando una entidad participa parcialmente de dos o mas relaciones.

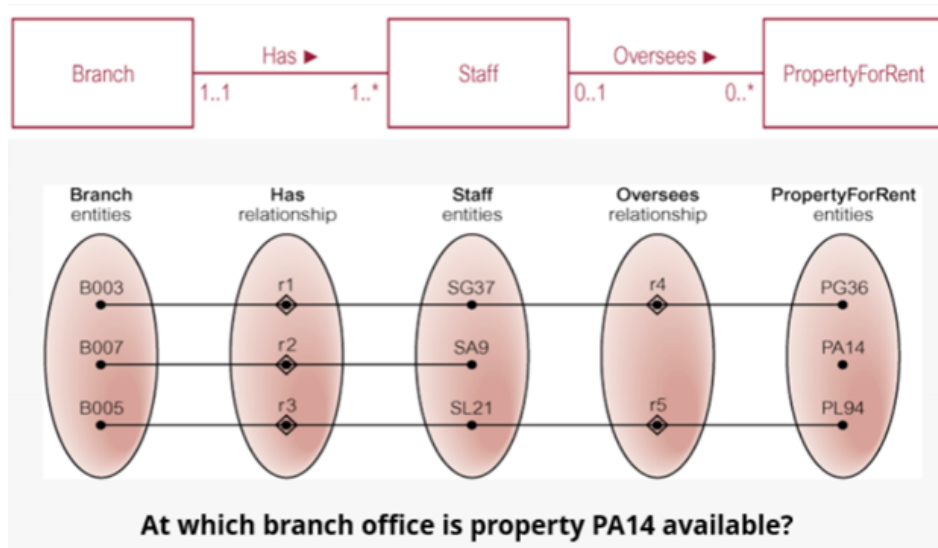


Figura 6: Trampa Sumidero Problema

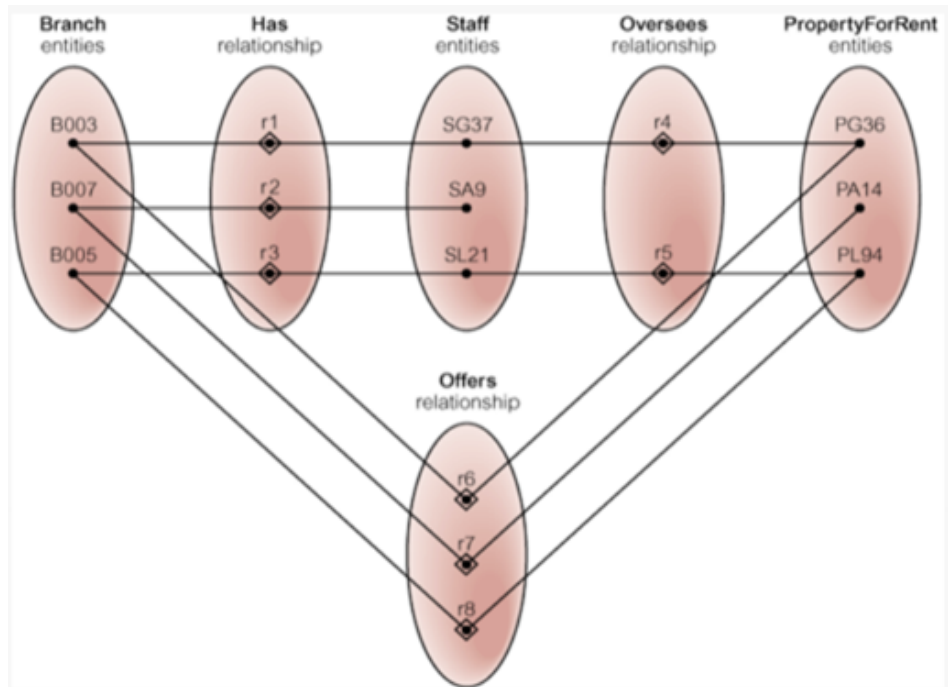


Figura 7: Trampa Sumidero Solución



## 3. Modelo Relacional

### Modelo de Tablas

Una base de datos relacional se organiza mediante tablas, también llamadas relaciones. Cada tabla consta de:

- **Columnas:** representan atributos de un dominio. Pueden ser tipos de datos elementales o complejos.
- **Filas o tuplas:** representan instancias de la entidad.
- **Claves primarias:** identifican de forma única cada fila.
- **Claves foráneas:** permiten establecer relaciones entre distintas tablas (1:1 o 1:N).
- **Constraints, vistas, triggers:** herramientas para definir reglas, vistas lógicas o reacciones automáticas.

### Heurísticas de diseño

- Si la PK es la misma, es la misma tabla.
- Evitar mezclar humanos con datos generados.
- Separar datos de metadatos.
- Evitar "bajas lógicas"(eliminaciones con flags).
- En reportes, evitar filtros salvo los imprescindibles.
- En claves compuestas, todos los campos (menos quizá el último) deben ser heredados.

### Clave Primaria

Debe ser:

- Ser **única**: no puede haber duplicados.
- **Existir siempre**: conocida desde antes de ingresar al sistema.
- **Ser estable**: no debe cambiar en el tiempo.
- **Preferentemente natural**: si es posible, un dato propio del dominio.
- **Menor cantidad de atributos**: conformada por la menor cantidad de atributos.

Usos de la clave primaria:

- Identificar registros dentro de la BD.
- Relacionar distintas entidades.
- Asociar con objetos externos (formularios, QR, móviles, sistemas distribuidos).

## Modelo Relacional en Bases de Datos

El modelo relacional está basado en la noción de relaciones matemáticas:

- Una **relación** es un conjunto de **tuplas** (filas) definidas sobre un conjunto de **dominios** (atributos).
- Cada tupla tiene  $n$  valores, uno por cada atributo del esquema de la relación.
- Formalmente, una relación  $S$  es un subconjunto del producto cartesiano  $D_1 \times D_2 \times \dots \times D_n$ :

$$S \subseteq D_1 \times D_2 \times \dots \times D_n$$

- Cada dominio  $D_i$  es un conjunto de valores válidos para el atributo  $i$ .
- Una tupla es entonces:  $(d_1, d_2, \dots, d_n)$  con  $d_i \in D_i$ .

## Operaciones Básicas

### Modelo Relacional - Operaciones Básicas

$$\sigma_{\varphi}(R)$$

$$\Pi_{a_1, \dots, a_n}(R)$$

$$\rho_{a/b}(R)$$

Figura 8: Operaciones básicas del modelo relacional

- **Selección:**  $\sigma_{\varphi}(R)$  devuelve las tuplas de  $R$  que cumplen la condición  $\varphi$ .
- **Proyección:**  $\Pi_{a_1, \dots, a_n}(R)$  devuelve la relación con solo los atributos especificados.
- **Renombramiento:**  $\rho_{a/b}(R)$  renombra el atributo  $b$  por  $a$  en la relación  $R$ .

## Otras Operaciones

- **Producto cartesiano:**  $R \times S$  combina todas las tuplas de  $R$  con todas las de  $S$ .
- **Join natural:**  $R \bowtie S$  une  $R$  y  $S$  por los atributos comunes, eliminando duplicados.
- **Theta join:**  $R \bowtie_{a\theta b} S$  une las tuplas de  $R$  y  $S$  que cumplen la condición  $a\theta b$ .
- **Semijoin izquierdo:**  $R \ltimes S$  conserva las tuplas de  $R$  que tienen coincidencia en  $S$ .

# Lenguaje SQL

El lenguaje SQL se divide en distintos subconjuntos, cada uno orientado a tareas específicas en el manejo de bases de datos.

- **DDL (Data Definition Language):** Permite **definir y modificar la estructura de la base de datos** (tablas, vistas, índices, esquemas). Opera sobre la **definición de objetos**, no sobre sus datos. Sus comandos más usados son:
  - **CREATE:** crea tablas, vistas, índices u otros objetos.
  - **ALTER:** modifica la estructura de un objeto existente (por ejemplo, agregar o eliminar columnas).
  - **DROP:** elimina permanentemente un objeto de la base de datos.

Notas: - Se pueden definir columnas con **tipos numéricos de punto fijo** (DECIMAL, NUMERIC) y de **punto flotante** (REAL, FLOAT). - No es obligatorio definir una clave primaria (PK), aunque es recomendado para evitar duplicados. - Las restricciones como PRIMARY KEY, FOREIGN KEY o NOT NULL también se declaran con DDL.

- **DML (Data Manipulation Language):** Permite **consultar y modificar los datos** dentro de las tablas. Sus comandos principales son:
  - **SELECT:** consulta registros; puede combinar datos de múltiples tablas mediante JOIN (puede devolver más filas que una sola tabla si se hacen combinaciones).
  - **INSERT:** inserta nuevos registros; puede combinarse con **SELECT** para copiar datos de otras tablas.
  - **UPDATE:** modifica registros existentes.
  - **DELETE:** elimina registros; puede incluir una condición **WHERE** para borrar selectivamente.
  - **TRUNCATE:** borra todos los registros de una tabla de forma más eficiente que **DELETE**, sin afectar su estructura.

Notas: - Las filas de una tabla (también llamadas **registros** o **tuplas**) pueden tener valores NULL. - Si no hay restricciones, pueden existir filas duplicadas. - Las operaciones sobre columnas que contengan NULL propagan el NULL (salvo que se usen funciones como COALESCE o agregaciones, que ignoran NULL por defecto).

- **DCL (Data Control Language):** Se usa para **gestionar permisos y seguridad** sobre los objetos de la base de datos. Comandos:
  - **GRANT:** otorga privilegios a un usuario o rol (por ejemplo, permiso de **SELECT** sobre una tabla).
  - **REVOKE:** revoca privilegios previamente concedidos.

Notas: - Permite aplicar el **principio del mínimo privilegio**, otorgando solo los permisos necesarios. - Suele usarse junto con mecanismos de autenticación a nivel de DBMS.

- **TCL (Transaction Control Language):** Controla las **transacciones**, garantizando las propiedades **ACID** (Atomicidad, Consistencia, Aislamiento y Durabilidad). Comandos:

- **BEGIN TRANSACTION:** inicia una transacción.
- **COMMIT:** confirma los cambios realizados dentro de la transacción.
- **ROLLBACK:** deshace los cambios si ocurre un error o si se decide abortar.

Notas: - Una transacción puede agrupar varias operaciones de **INSERT**, **UPDATE**, **DELETE**.  
- Los distintos niveles de aislamiento (**READ COMMITTED**, **REPEATABLE READ**, **SERIALIZABLE**) controlan la concurrencia y evitan anomalías como *lost updates* o *incorrect summaries*.  
- El nivel **SERIALIZABLE** garantiza que el resultado sea equivalente a una ejecución en serie (aunque puede generar bloqueos y deadlocks).

## Claves primarias en SQL

En SQL, una tabla **no está obligada** a tener clave primaria, aunque es altamente recomendable para mantener integridad y facilitar búsquedas. Si no hay PK:

- El sistema no garantiza unicidad de filas.
- Motores como InnoDB (MySQL) crean internamente un índice único oculto para optimizar accesos.

## Claves foráneas (FK)

Las claves foráneas deben apuntar a columnas que sean:

- Una **clave primaria (PK)**, o
- Una **clave única (UNIQUE)** en la tabla referenciada.

No pueden referenciar columnas sin restricción de unicidad, ya que se perdería la integridad referencial (podría haber múltiples coincidencias).

## Tratamiento de datos faltantes (NULL)

- Un atributo puede ser NULL cuando:
  - El dato es desconocido.
  - No corresponde por otros atributos (ej. sueldo de ad honorem).
  - No clasificado.
  - Error de representación.
- **Restricciones:**
  - Claves primarias no permiten NULL.
  - Claves únicas sí, como claves parciales.
  - Foráneas pueden ser NULL (depende de configuración MATCH).

## Semántica de NULL en SQL

En SQL, la lógica es **trivaluada** (TRUE, FALSE, NULL)

Las principales reglas de evaluación son:

- $a + \text{NULL} \implies \text{NULL}$  (cualquier suma que incluya un valor desconocido produce un resultado desconocido).
- $0 * \text{NULL} \implies \text{NULL}$  (aunque 0 multiplicado por cualquier valor numérico da 0, al ser NULL un valor desconocido, el resultado también es desconocido, ya que no se sabe si el valor es finito o válido).
- $a = \text{NULL}$  o  $\text{NULL} \neq a \implies \text{NULL}$  (no se puede afirmar que  $a$  sea igual o distinto a un valor desconocido).
- $\text{NULL} = \text{NULL} \implies \text{NULL}$  (dos valores desconocidos no son considerados iguales, ya que no se conoce su valor).
- $\text{NULL} || ' ' || \text{'apellido'}$   $\implies \text{NULL}$  (la concatenación con un valor desconocido produce un resultado desconocido).
- $\text{NULL IS NULL} \implies \text{TRUE}$  (verifica explícitamente si el valor es NULL).
- $\text{NULL IS NOT NULL} \implies \text{FALSE}$  (confirma que no es un valor válido conocido).

null equivale a desconocido en expresiones lógicas

| AND   | true  | false | null  |
|-------|-------|-------|-------|
| true  | true  | false | null  |
| false | false | false | false |
| null  | null  | false | null  |

| OR    | true | false | null |
|-------|------|-------|------|
| true  | true | true  | true |
| false | true | false | null |
| null  | true | null  | null |

| =     | true  | false | null |
|-------|-------|-------|------|
| true  | true  | false | null |
| false | false | true  | null |
| null  | null  | null  | null |

| IS    | true  | false | null  |
|-------|-------|-------|-------|
| true  | true  | false | false |
| false | false | true  | false |
| null  | false | false | true  |

salvo para las siguientes operaciones:

- IS (= entre bool)
- IS DISTINCT FROM ( $\neq$  any type)
- CONCAT
- todas las funciones de agregación: SUM, AVG, MIN, MAX

| NOT | true  | false | null |
|-----|-------|-------|------|
|     | false | true  | null |

Figura 9: Lógica de 3 estados

## Comportamiento de NULL en Concatenaciones y Funciones de Agregación

### Concatenaciones (CONCAT y ||)

- Si cualquiera de los operandos de || es NULL, el resultado completo también es NULL.
- La función CONCAT() omite los valores NULL si hay otros valores presentes; si todos son NULL, devuelve NULL

### Funciones de Agregación

- Funciones como SUM(), AVG(), MIN() y MAX() ignoran los valores NULL (los tratan como inexistentes)
- COUNT(\*) cuenta todas las filas, incluidas las que tienen NULL. COUNT(columna) no cuenta las filas donde la columna es NULL:
- Si todos los valores de la columna son NULL, las funciones de agregación (excepto COUNT(\*)) devuelven NULL.

## Manejo de NULL en SQL

En SQL, el valor NULL representa un dato **desconocido** o **indefinido**. Aunque no puede convertirse directamente en un valor, existen funciones y técnicas para evitar que afecte cálculos, concatenaciones o comparaciones.

### Funciones para reemplazar NULL

- **COALESCE(expr1, expr2, ..., exprN)**: Devuelve el primer valor que no sea NULL. Es estándar en SQL.

```
SELECT COALESCE(nombre, 'Desconocido') AS nombre_mostrado
FROM Clientes;
```

- **IFNULL(expr, valor)**: Función específica de MySQL que devuelve valor si expr es NULL.

### Control con CASE

Permite decidir explícitamente qué devolver cuando un valor sea NULL:

```
SELECT CASE
    WHEN precio IS NULL THEN 0
    ELSE precio
END AS precio_final
FROM Productos;
```

## 4. Normalización de Bases de Datos

### Objetivo de la Normalización

La normalización es el proceso de organizar los datos de una base de datos para:

- Eliminar redundancias.
- Evitar anomalías de inserción, actualización y borrado.
- Facilitar el mantenimiento de la integridad de los datos.

### Conceptos clave:

- **Atributo atómico:** no se puede descomponer en otros atributos más simples. Ej: DNI.
- **Atributo compuesto:** formado por más de un atributo primitivo. Ej: nombre completo = nombre + apellido.
- **Atributo derivado:** se calcula a partir de otros datos. Ej: edad a partir de fecha de nacimiento.
- **Atributo multivaluado:** puede tener varios valores. Ej: teléfonos de un cliente.
- **Atributo clave:** atributo o conjunto de atributos que identifican unívocamente una entidad.
- **Atributo primo:** atributo que forma parte de alguna clave candidata.
- **Atributo no primo:** no forma parte de ninguna clave candidata.
- **Dependencia funcional:**  $X \rightarrow Y$  indica que el valor de  $X$  determina unívocamente el de  $Y$ .

**Tipos de Dependencias Funcionales:** Dada una dependencia  $X \rightarrow Y$  es

- **Parcial:** Existe un subconjunto  $Z \subset X$  tal que  $Z \rightarrow Y$ . Es decir, no se necesitan todos los atributos de  $X$  para determinar  $Y$ .
- **Completa:** Ningún subconjunto propio de  $X$  determina  $Y$ .
- **Transitiva:** Existe un conjunto de atributos  $Z$  tal que:  $X \rightarrow Z$  y  $Z \rightarrow Y$ . De esta forma,  $X$  determina a  $Y$  a través de  $Z$ .
- **Superclave:** conjunto de atributos que identifica de forma unívoca a cada tupla.
- **Clave candidata:** superclave minimal, es decir, que no tiene atributos redundantes.
- **Clave primaria:** clave candidata elegida para identificar las tuplas.

## Ejemplo: Relación FACULTAD

Supongamos que queremos registrar para una facultad los datos personales de los alumnos, las materias en las que se inscribieron y los exámenes que rindieron.

FACULTAD(LU, NOMBRE, MATERIA, IFEC, EFEC, NOTA)

Donde:

- **IFEC** es la fecha de inscripción
- **EFEC** es la fecha del examen

Y el conjunto de dependencias funcionales  $F$  es:

- $LU \rightarrow NOMBRE$  (no puede haber dos alumnos con el mismo LU)
- $LU, MATERIA \rightarrow IFEC$  (se puede inscribir una sola vez en cada materia)
- $LU, MATERIA, EFEC \rightarrow NOTA$  (hay una sola nota por examen)

Nota: Como no se cumple  $LU, MATERIA \rightarrow EFEC$ , se permite rendir varias veces la misma materia.

**Inferencia:** Decimos que un conjunto de dependencias funcionales  $F$  **infiere** una dependencia  $f$ , lo cual se denota como  $F \models f$ , si toda relación que satisface  $F$  también satisface  $f$ .

## Axiomas de Armstrong y Reglas de Inferencia

- **Reflexividad:** Si  $Y \subseteq X$ , entonces  $X \rightarrow Y$ .
- **Aumento:** Si  $X \rightarrow Y$ , entonces  $XZ \rightarrow YZ$  para cualquier  $Z$ .
- **Transitividad:** Si  $X \rightarrow Y$  y  $Y \rightarrow Z$ , entonces  $X \rightarrow Z$ .

Reglas derivadas:

- **Unión:** Si  $X \rightarrow Y$  y  $X \rightarrow Z$ , entonces  $X \rightarrow YZ$ .
- **Descomposición:** Si  $X \rightarrow YZ$ , entonces  $X \rightarrow Y$  y  $X \rightarrow Z$ .
- **Pseudotransitividad:** Si  $X \rightarrow Y$  y  $YW \rightarrow Z$ , entonces  $XW \rightarrow Z$ .



**Clausura:** La clausura de  $X$  respecto de  $F$ , denotada como  $X^+$ , es el conjunto de todos los atributos que pueden derivarse funcionalmente de  $X$  usando las reglas de inferencia.

Para calcular  $X^+$  :

1.  $X_0 = X$ .
2.  $X_{i+1} = X_i \cup \{A \mid \exists(Y \rightarrow Z) \in F, A \in Z, Y \subseteq X_i\}$  (Conjunto de atributos  $A$  tal que hay alguna dependencia funcional  $Y \rightarrow Z$  en  $F$ ,  $A$  está en  $Z$  e  $Y$  está incluida en  $X_i$ )
3. Repetir el paso (2) hasta que  $X_i = X_{i+1}$  (punto fijo).

$$F^+ = \{X \rightarrow Y / F \models X \rightarrow Y\}$$

### Ejemplo 3

$R(A, B)$

$F : A \rightarrow B$

$F^+ = \{A \rightarrow B, A \rightarrow A, B \rightarrow B, AB \rightarrow A, AB \rightarrow B, AB \rightarrow AB, A \rightarrow AB\}$

### Ejemplo 4

$R(A, B, C)$

$F = \{AB \rightarrow C, C \rightarrow BB\}$

$F^+ = \{A \rightarrow A, AB \rightarrow A, AC \rightarrow A, ABC \rightarrow A, B \rightarrow B, AB \rightarrow B, BC \rightarrow B, ABC \rightarrow B, C \rightarrow C, AC \rightarrow C, BC \rightarrow C, ABC \rightarrow C, AB \rightarrow AB, ABC \rightarrow AB, AC \rightarrow AC, ABC \rightarrow AC, BC \rightarrow BC, ABC \rightarrow BC, ABC \rightarrow ABC, AB \rightarrow C, AB \rightarrow AC, AB \rightarrow BC, AB \rightarrow ABC, C \rightarrow B, C \rightarrow BC, AC \rightarrow B, AC \rightarrow AB\}$

## Equivalencia de Conjuntos de Dependencias Funcionales

Decimos que dos conjuntos de dependencias funcionales  $F$  y  $G$  sobre un esquema de relación  $R$  son **equivalentes** (denotado  $F \equiv G$ ) si cumplen alguna de las siguientes condiciones equivalentes:

$$F^+ = G^+,$$

es decir, si ambos conjuntos implican exactamente las mismas dependencias funcionales. De forma equivalente, también se cumple que:

$$F \models G \quad \text{y} \quad G \models F,$$

**Cubrimiento minimal:** Es un conjunto equivalente de dependencias funcionales  $F$  que cumple:

- El lado derecho de cada dependencia es un solo atributo.
- No hay atributos redundantes en el lado izquierdo de ninguna dependencia.
- Ninguna dependencia es redundante.

## Descomposición con Pérdida de Información

Al descomponer una relación  $R$  en subrelaciones  $\rho = \{R_1, R_2, \dots, R_k\}$ , puede producirse **pérdida de información** si, al reconstruir  $R$  mediante joins, aparecen **tuplas espurias** que no existían en los datos originales.

Una descomposición es **sin pérdida de información** (*lossless join* o SPI) cuando, al proyectar cualquier instancia  $r$  sobre cada  $R_i$  y unir las nuevamente, se recupera **exactamente**  $r$ , sin generar datos adicionales.

### Condición para Descomposición Binaria

Sea una descomposición  $\rho = \{R_1, R_2\}$  de  $R$ . La descomposición es **lossless join** respecto de  $F$  si y solo si se cumple alguna de las siguientes condiciones:

$$(R_1 \cap R_2) \rightarrow (R_1 - R_2) \in F^+$$

o

$$(R_1 \cap R_2) \rightarrow (R_2 - R_1) \in F^+.$$

En otras palabras, la intersección  $R_1 \cap R_2$  debe ser una **superclave** para al menos una de las dos proyecciones  $R_1$  o  $R_2$ .

## Algoritmo de Tableau para Verificar Pérdida de Información

La regla de la descomposición binaria aplica sólo cuando una relación se divide en dos subesquemas. Para casos generales, se usa el **algoritmo de Tableau**:

- Dados  $R$ ,  $F$  y  $\rho = (R_1, R_2, \dots, R_k)$ , se construye un **tableau inicial**  $T_0$ :
  - Las columnas corresponden a los atributos de  $R$ .
  - Cada fila representa un subesquema  $R_i$ : los atributos presentes en  $R_i$  se marcan con **símbolos distinguidos**  $a_j$ , los ausentes con **símbolos no distinguidos**  $b_{ij}$ .
- Se aplican las dependencias funcionales  $X \rightarrow A$ : si dos filas tienen los mismos valores para  $X$  pero difieren en  $A$ , se unifican:
  1. Si uno de los valores es distinguido  $a_j$ , ambos toman  $a_j$ .
  2. Si ambos son no distinguidos  $b_{ij}$  y  $b_{hj}$ , se reemplaza uno por el otro.

- Se repite hasta alcanzar un tableau final  $T^*$  donde no se puedan hacer más cambios.
- **Criterio:** Si  $T^*$  contiene una fila con **todos símbolos distinguidos**, la descomposición  $\rho$  es **sin pérdida de información (SPI)**. De lo contrario, la descomposición tiene pérdida de información.

### Ejemplo 10

$$R = (A, B, C, D)$$

$$F = \{A \rightarrow B, AC \rightarrow D\}$$

$$\rho = \{(A, B), (A, C, C)\}$$

Hacemos  $T_0$

|     | A     | B        | C        | D        |
|-----|-------|----------|----------|----------|
| AB  | $a_1$ | $a_2$    | $b_{13}$ | $b_{14}$ |
| ACD | $a_1$ | $b_{22}$ | $a_3$    | $a_4$    |

Usamos la dependencia  $A \rightarrow B$  por lo que tendríamos que reemplazar  $b_{22}$  con  $a_2$  lo cual nos lleva al tableau  $T_1$

|     | A     | B     | C        | D        |
|-----|-------|-------|----------|----------|
| AB  | $a_1$ | $a_2$ | $b_{13}$ | $b_{14}$ |
| ACD | $a_1$ | $a_2$ | $a_3$    | $a_4$    |

Figura 10: Cómo la segunda fila tienen todos elementos distinguidos podemos concluir que es SPI.

## Pérdida de Dependencias Funcionales

Una descomposición  $\rho$  puede **preservar las dependencias funcionales** del esquema original, lo que se denomina **descomposición sin pérdida de dependencias funcionales (SPDF)**.

### Proyección de un conjunto de dependencias funcionales

La **proyección** de un conjunto de dependencias funcionales  $F$  sobre un conjunto de atributos  $Z$ , denotada como  $\pi_Z(F)$ , es el conjunto de dependencias  $X \rightarrow Y \in F^+$  tal que  $XY \subseteq Z$ .

## Criterio de preservación

Dada una relación  $R$ , un conjunto de dependencias  $F$  y una descomposición  $\rho = (R_1, R_2, \dots, R_k)$ , decimos que  $\rho$  **preserva**  $F$  si:

$$F^+ = \left( \bigcup_{i=1}^k \pi_{R_i}(F) \right)^+$$

Es decir, si las dependencias funcionales que resultan de proyectar  $F$  sobre cada  $R_i$  (y luego unirlos) implican lógicamente a  $F$ .

## Algoritmo para Verificar Pérdida de Dependencias (SPDF)

Para determinar si una descomposición  $\rho = (R_1, R_2, \dots, R_k)$  preserva las dependencias funcionales de un conjunto  $F$ , sin calcular explícitamente  $F^+$ , se utiliza el siguiente algoritmo.

Queremos verificar, para cada dependencia  $X \rightarrow Y$ , si se preserva.

1. Inicializar  $Z := X$ .
2. Mientras  $Z$  cambie, hacer:

a) Para cada subesquema  $R_i$  de la descomposición:

$$Z := Z \cup ((Z \cap R_i)^+ \cap R_i)$$

donde la clausura  $(Z \cap R_i)^+$  se calcula con respecto a  $F$ .

3. Terminar cuando  $Z$  no cambie más.
4. Si  $Y \subseteq Z$ , entonces  $X \rightarrow Y$  está en  $F^+$  y la dependencia se preserva.

El algoritmo puede detenerse tempranamente si  $Y \subseteq Z$  se cumple durante alguna iteración.

Si este resultado se cumple para **todas las dependencias**  $X \rightarrow Y$  en  $F$ , entonces  $\rho$  es una descomposición **sin pérdida de dependencias funcionales (SPDF)**.

### Cálculo de $Z := Z \cup ((Z \cap R_i)^+ \cap R_i)$

1. **Intersección con el subesquema:** Tomamos  $Z \cap R_i$ , es decir, filtramos de  $Z$  solo los atributos que pertenecen al subesquema  $R_i$ .
2. **Clausura con respecto a  $F$ :** Calculamos la clausura  $(Z \cap R_i)^+$  respecto de  $F$ , obteniendo todos los atributos que se pueden deducir a partir de esos atributos utilizando las dependencias funcionales.
3. **Restricción al subesquema:** Intersectamos la clausura con  $R_i$ , es decir, tomamos  $(Z \cap R_i)^+ \cap R_i$ . Esto conserva únicamente los atributos de la clausura que están en el subesquema  $R_i$ .
4. **Actualización de  $Z$ :** Finalmente, unimos estos atributos a  $Z$  mediante:

$$Z := Z \cup ((Z \cap R_i)^+ \cap R_i).$$

De esta forma,  $Z$  se expande progresivamente con todos los atributos deducibles dentro de cada  $R_i$ .

## Anomalías y Descomposición

- **Redundancia de información:** el nombre del alumno se repite por cada materia en la que se inscribe y por cada examen que rinde.
- **Anomalía de actualización:** si un alumno cambia su nombre, debe modificarse en varias tuplas. Esto puede generar inconsistencias o errores si alguna tupla no se actualiza correctamente.
- **Anomalía de inserción:** no se puede registrar un nuevo alumno si no se inscribió a una materia o si no rindió ningún examen. Se necesitarían valores nulos incluso en atributos que forman parte de la clave. Ejemplos:

```
FACULTAD(lu1, nom1, -, -, -, -)          % alumno sin inscripciones
FACULTAD(-, -, -, mat1, -, efec1, -)      % examen sin alumnos
```

- **Anomalía de borrado:** si se eliminan los datos de los exámenes de un alumno, podrían perderse también sus datos personales.

**Descomposición:** Descomponer una relación es dividir el esquema en varios subesquemas de tal manera que **todos los atributos del esquema estén presentes en al menos un subesquema**.

## Formas Normales

- **Primera Forma Normal (1FN):** Una relación está en 1FN si todos los atributos contienen valores atómicos.
- **Segunda Forma Normal (2FN):** R está en 2FN si todo atributo no primo A en R es totalmente dependiente de todas las claves de R.
- **Tercera Forma Normal (3FN):** R está en tercera forma normal (3FN) si para toda dependencia funcional no trivial  $X \rightarrow A$  sobre R, se cumple que
  1. X es superclave de R o
  2. A es primo
- **Forma Normal de Boyce-Codd (FNBC):** R está en FNBC si para toda dependencia funcional no trivial  $X \rightarrow A$  sobre R, X es una superclave de R

Todo esquema de dos atributos  $R = (A, B)$  está en FNBC, ya que:

1. Si A es clave, entonces  $A \rightarrow B$  y  $A \rightarrow AB$ . Si además  $B \rightarrow A$ , entonces B también es clave candidata ( $B \rightarrow AB$ ). En ambos casos, las dependencias cumplen con FNBC.
2. Si B es clave, se aplica el mismo razonamiento que para A.
3. Si AB es clave, entonces  $AB \rightarrow AB$ , por lo que AB es superclave y se cumple la FNBC.

## Algoritmo de Descomposición SPI en FNBC

- Sea una relación  $R$  con dependencia  $X \rightarrow Y$  que viola BCNF.
- Se reemplaza  $R$  por:
  - $R_1 = XY$
  - $R_2 = R - Y$

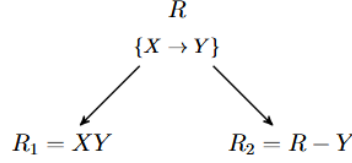
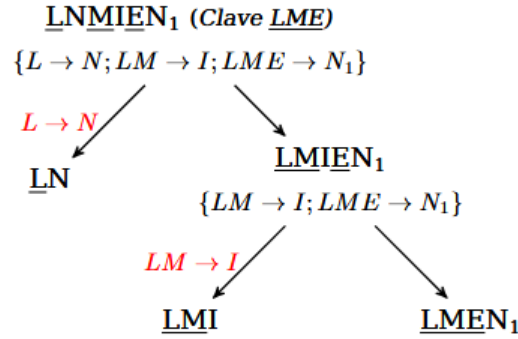


Figura 11: Esquema de descomposición por dependencia

## Ejemplo Descomposición BCNF



Finalmente  $\rho = \{(L, N), (L, M, I), (L, M, E, N_1)\}$  está en FNBC y es SPI

Figura 12: Descomposición paso a paso

**Resultado:**  $\rho = \{(L, N), (L, M, I), (L, M, E, N_1)\}$  está en BCNF y es SPI.

## Algoritmo de Descomposición en 3FN (SPI y SPDF)

Este algoritmo construye una descomposición  $\rho$  de una relación  $R$ , garantizando que sea:

**sin pérdida de información (SPI) y sin pérdida de dependencias (SPDF).**

Parte de una **cobertura minimal**  $F_M$  del conjunto de dependencias funcionales  $F$ .

1. Para cada dependencia funcional  $X \rightarrow A$  en  $F_M$ , crear un subesquema  $(X, A)$ .
2. Unificar subesquemas que tengan el mismo lado izquierdo:

$$(X, A_1), (X, A_2), \dots \rightarrow (X, A_1, A_2, \dots, A_n).$$

3. Si ningún subesquema contiene una **clave candidata** de  $R$ , agregar un esquema que contenga los atributos de alguna clave.
4. Eliminar esquemas redundantes: si un subesquema está contenido por completo en otro, descártalo.

El resultado  $\rho$  es una descomposición en Tercera Forma Normal que preserva información y dependencias.

## 5. Lenguajes de Consulta

### Introducción

En esta sección se presentan los principales lenguajes de consulta sobre bases de datos relacionales. Se enfocan tanto en la **descripción del resultado** (lenguajes declarativos) como en el **modo de obtenerlo** (lenguajes procedimentales).

### Clasificación

- **Lenguajes de Bajo Nivel:** especifican **cómo** obtener el resultado. Ejemplo: *Álgebra Relacional*.
- **Lenguajes de Alto Nivel:** especifican **qué** resultado se quiere. Ejemplos: *Cálculo Relacional* (de tuplas o dominios), *SQL*.

### Álgebra Relacional (AR)

**Definición:** lenguaje de consulta procedimental. Cada consulta devuelve una nueva relación y se construye mediante operadores que actúan sobre una o más relaciones de entrada.

Las operaciones de AR pueden componerse. El input de una operación puede ser el resultado de una operación.

#### Operadores básicos:

- **Proyección** ( $\pi_{A_1, \dots, A_n}(R)$ ): devuelve una relación con solo los atributos especificados, eliminando duplicados.
- **Selección** ( $\sigma_{cond}(R)$ ): devuelve las tuplas de  $R$  que cumplen la condición  $cond$ .
- **Unión** ( $R \cup S$ ), **Intersección** ( $R \cap S$ ), **Diferencia** ( $R - S$ ): combinan relación por conjunto.
- **Producto Cartesiano** ( $R \times S$ ): combina todas las tuplas de  $R$  con todas las de  $S$ .
- **Join** ( $R \bowtie_{cond} S$ ): combina tuplas de  $R$  y  $S$  que cumplen cierta condición.
- **Join Natural** ( $R \bowtie S$ ): join sobre atributos comunes con eliminación de duplicados.
- **Join Externo (Izquierdo, Derecho, Completo)**: conservan las tuplas sin correspondencia de una o ambas relaciones.
- **División** ( $R(Z) \div S(X)$ ): devuelve tuplas  $Y = Z - X$  tales que combinan con todas las tuplas de  $S$ .
- **Renombrado** ( $\rho_{a_1 \rightarrow a_2}(R)$ ): cambia el nombre de atributos o relaciones.



## Propiedades del Álgebra Relacional

Las consultas en AR **no contienen duplicados**. Todas las operaciones producen nuevas relaciones.

La **selectividad** de un predicado  $p$  es la proporción de tuplas de una relación que satisfacen el predicado.

### Propiedades de la Selección

1. **Cascada:** Una selección compuesta puede expresarse como una sola selección con condiciones combinadas por AND.

$$\sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_1 \wedge C_2}(R)$$

2. **Conmutatividad:** El orden en que se aplican las selecciones no altera el resultado:

$$\sigma_{C_1}(\sigma_{C_2}(R)) = \sigma_{C_2}(\sigma_{C_1}(R))$$

### Propiedades de la Proyección

1. **Cascada:** Dos proyecciones en cascada pueden reducirse a una sola, tomando la intersección de atributos:

$$\pi_{A_1, \dots, A_n}(\pi_{B_1, \dots, B_m}(R)) = \pi_{A_1, \dots, A_n}(R), \quad \text{donde } \{A_i\} \subseteq \{B_j\}.$$

2. **Conmutatividad con Selección:** Una proyección y una selección pueden intercambiarse, siempre que la proyección incluya los atributos usados en la condición de la selección:

$$\pi_{A_1, \dots, A_n}(\sigma_C(R)) = \sigma_C(\pi_{A_1, \dots, A_n}(R)),$$

si los atributos de  $C$  están contenidos en  $\{A_1, \dots, A_n\}$ .

### Propiedades del Join (Producto Cartesiano)

1. **Conmutatividad:** El producto cartesiano (y por extensión el join) es conmutativo:

$$R \times S = S \times R$$

(aunque cambia el orden de los atributos).

2. **Conmutatividad respecto de la Selección:** Una selección puede aplicarse antes o después del join:

$$\sigma_C(R \times S) = (\sigma_C(R)) \times S,$$

siempre que  $C$  involucre solo atributos de  $R$ .

## Heurísticas de Optimización de Consultas

1. **Ejecutar primero los joins más selectivos:** Esto permite reducir la cantidad de tuplas que se propagan hacia las operaciones posteriores, disminuyendo el tamaño intermedio de los resultados.
2. **Mover proyecciones y selecciones lo más cerca posible de las hojas del árbol de consulta:** De esta forma, se reducen las tuplas y atributos procesados por operaciones posteriores. Esta estrategia debe aplicarse siempre que:
  - No interfiera con otras operaciones (por ejemplo, manteniendo atributos necesarios para futuros joins).
  - No elimine índices que puedan mejorar el rendimiento del plan.
3. **Reemplazar productos cartesianos seguidos de selecciones por joins:** Esto evita el crecimiento exponencial del tamaño intermedio, reduciendo la cantidad de tuplas que se deben manejar.

**Importancia:** La AR es la base para entender los procesos internos de los sistemas gestores de bases de datos y sirve como paso previo a la optimización y traducción a SQL.

## Cálculo Relacional de Tuplas (CRT)

**Definición:** Es un lenguaje formal de consultas para recuperar datos almacenados en un modelo relacional. Lenguaje de consulta declarativo basado en el cálculo de predicados. Una consulta se describe como una **fórmula lógica** que especifica las condiciones que deben cumplir las tuplas del resultado. **Sintaxis general:**

$$\{ t \mid P(t) \}$$

Donde  $t$  es una variable de tupla y  $P(t)$  es una fórmula lógica que puede incluir:

- Expresiones atómicas:  $t.A = c$ ,  $t.A = t'.B$ ,  $R(t)$  (la tupla  $t$  pertenece a la relación  $R$ ).
- Conectivos lógicos:  $\wedge$ ,  $\vee$ ,  $\neg$ .
- Cuantificadores:  $\exists$ ,  $\forall$ .

### Ventajas:

- Lenguaje más cercano a la lógica y a los usuarios con formación formal.
- Permite describir condiciones complejas de forma compacta.

## Expresividad de los lenguajes de consulta

- AR y CRT tienen el mismo poder de expresión que la lógica de primer orden (para consultas seguras).
- SQL tiene al menos el mismo poder de expresión, con extensiones: aritmética compleja, recursión, agregación, funciones almacenadas.

## 6. Control de Concurrency

### A.C.I.D.: Propiedades de las Bases de Datos Transaccionales

En los sistemas de control de concurrencia y recovery procuramos que se cumplan una serie de propiedades:

- **Atomicidad:** una operación compuesta se ejecuta completamente o no se ejecuta nada.
- **Consistencia:** las lecturas retornan siempre el valor más reciente del ítem leído.
- **Aislamiento:** las operaciones concurrentes no interfieren entre sí.
- **Durabilidad:** los cambios confirmados persisten ante fallos del sistema.

### Introducción

Cuatro problemas clásicos del control de concurrencia:

- **Lost update:** dos transacciones escriben sobre el mismo dato sin ver los cambios del otro, sobrescribiendo información sin intención. T2 lee X previo a que T1 lo modifique y T2 sobrescribe el valor sin ver la escritura de T1, faltando una actualización de X.
- **Dirty read:** una transacción lee datos que otra transacción modificó pero que aún no confirmó. Si esta última aborta, la primera queda con datos inconsistentes.
- **Non-repeatable read:** una transacción lee un dato, luego otra lo modifica y lo confirma. Si la primera vuelve a leer, obtiene un valor distinto.
- **Incorrect summary:** Ocurre cuando una transacción T1 está actualizando valores que otra transacción T2 está utilizando para calcular una función de agregación de suma. Algunos valores serán modificados antes de ser sumados y otros no, obteniendo un valor incorrecto.
- **Phantom problem:** una transacción ejecuta una consulta con una condición y otra transacción inserta un nuevo registro que cumple la condición. La primera transacción reejecuta la misma consulta y obtiene un resultado diferente, aunque no modificó directamente los datos existentes.

#### Soluciones:

- **Lockeo de predicados:** consiste en bloquear la condición lógica del WHERE. Muy costoso computacionalmente y difícil de implementar (problema NP-hard).
- **Index Locking:** se bloquean las hojas del índice utilizado por la consulta. Esto evita que se inserten o borren registros que cambiarían el resultado.
- **Nivel Serializable:** previene los phantoms mediante estrategias como validación por rangos o ejecución secuencial.

## Niveles de Aislamiento

- **Read Uncommitted:** Permite que las lecturas accedan a versiones no confirmadas.
- **Read Committed:** Solo permite leer datos confirmados por otras transacciones.
- **Repeatable Read:** Asegura que si una transacción lee el mismo valor dos veces, no cambiará.
- **Serializable:** Se comporta como si las transacciones se ejecutaran una tras otra, en orden.

| NIVELES DE AISLAMIENTO EN SQL |              |             |                      |             |
|-------------------------------|--------------|-------------|----------------------|-------------|
| Isolation level               | Lost updates | Dirty reads | Non-repeatable reads | Phantoms    |
| Read Uncommitted              | don't occur  | may occur   | may occur            | may occur   |
| Read Committed                | don't occur  | don't occur | may occur            | may occur   |
| Repeatable Read               | don't occur  | don't occur | don't occur          | may occur   |
| Serializable                  | don't occur  | don't occur | don't occur          | don't occur |

Figura 13: Niveles de Aislamiento en SQL

## Concurrencia pesimista

El **enfoque pesimista** asume que las transacciones pueden entrar en conflicto, por lo tanto, protege los recursos usando **mecanismos de bloqueo**.

## Historia y propiedades de correctitud

- **Historia:** es un orden parcial sobre un conjunto de operaciones que pertenecen a un conjunto de transacciones.
- **Operación conflictiva:** Dos operaciones están en conflicto con respecto a un dato, si ambas operan sobre el mismo dato y al menos una de ellas escribe.
- **Equivalencia por conflictos:** dos historias son conflicto equivalentes si están definidas sobre el mismo conjunto de transacciones y el orden de las operaciones conflictivas de las transacciones que no abortan es el mismo.

- **Historia serial:** Dado un par de transacciones  $T_i, T_j$ , y sus operaciones, una historia es serial si todas las operaciones de  $T_i$  se ejecutan antes de  $T_j$  (en la historia) o viceversa.
- **Historia serializable por conflictos (SR):** es aquella equivalente a una historia serial.
- **Y lee de X:** Una transacción  $Y$  lee un valor  $C$  de otra transacción  $X$  cuando  $X$  fue la última transacción en escribir  $X$  antes de que  $Y$  la leyera y no abortó antes de que la leyera.

#### Grafo de precedencia:

- Nodo por cada transacción.
- Arco  $T_i \rightarrow T_j$  si existe un par de operaciones conflictivas  $op_i < op_j$  sobre el mismo dato.
- $H$  es serializable por conflictos  $\iff$  el grafo es acíclico.
- Cada orden topológico del grafo corresponde a un orden serial equivalente.

### Recuperabilidad y variantes

- **Historia recuperable (RC):** si  $T_i$  lee un valor escrito por  $T_j$  ( $w_j(x) < r_i(x)$ ), entonces  $c_j < c_i$ .
- **Avoids Cascading Aborts (ACA):** si  $T_i$  lee de  $T_j$  ( $w_j(x) < r_i(x)$ ), entonces  $c_j < r_i$ .
- **Strict (ST):** si  $w_j(x) < o_i(x)$  implica  $c_j < o_i(x)$  o  $a_j < o_i(x)$ . Es decir, **toda operación** de lectura o escritura sobre un dato  $x$  modificado por  $T_j$  solo puede ejecutarse después del commit o abort de  $T_j$ :

#### Contención:

$$\text{ST} \subset \text{ACA} \subset \text{RC}$$

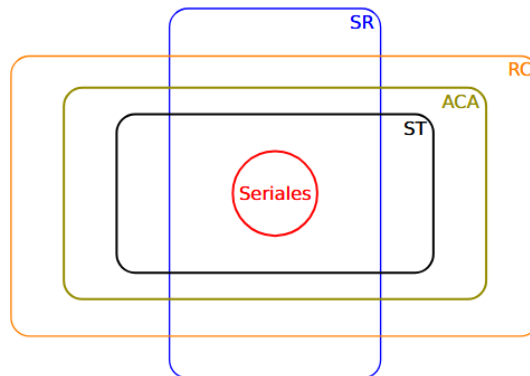


Figura 14: Relación entre recuperabilidad y serializabilidad

**SR** intersecta a todos los conjuntos **RC**, **ACA** y **ST**. Son conceptos ortogonales. Es fácil ver que una **historia serial** es también **ST**.

## Control de concurrencia: Lock Binario

- **Operaciones:**

- $l_i(x)$ :  $T_i$  bloquea  $x$
- $u_i(x)$ :  $T_i$  libera el lock

- **Grafo de precedencia:** si  $l_i(x) < l_j(x)$ , entonces  $T_i \rightarrow T_j$

- **Legalidad:** Para poder tomar un lock sobre  $x$ , ningún otro lock activo puede interferir (hay que esperar unlock).

## Lock Ternario (Shared/Exclusive)

- **Locks:**

- $rl_i(x)$ : lock compartido (lectura)
- $wl_i(x)$ : lock exclusivo (escritura)

- **Grafo de precedencia:**

- $ol_i(x) < wl_j(x) \Rightarrow T_i \rightarrow T_j$
- $wl_i(x) < rl_j(x) \Rightarrow T_i \rightarrow T_j$

- **Legalidad:**

- Si  $wl_i(x)$  está activo, ningún otro lock puede operar sobre  $x$  sin que  $u_i(x)$  ocurra antes.
- Si  $rl_i(x)$  está activo, ningún  $wl_j(x)$  puede obtenerse sin que  $u_i(x)$  ocurra antes.

## Upgrading y Update Lock

- **Upgrading:** convertir un lock compartido en exclusivo.
- **Update Lock (UL):** lock especial que permite leer y luego convertir a exclusivo. Denotado  $ul_i(x)$ .
- **Ventaja:** solo una transacción puede tener  $ul_i(x)$  sobre  $x$ , evitando conflictos al intentar upgrade.

## Two Phase Locking (2PL)

- Una transacción cumple **2PL** si todas las operaciones de adquisición de locks ocurren antes de la primera liberación.
- Fase creciente: adquisición de locks.
- Fase decreciente: liberación de locks.
- Si todas las transacciones cumplen 2PL, la historia es serializable por conflictos (**SR**).

## 2PL Estricto (Strict 2PL)

- Es **2PL** y los **locks de escritura** se mantienen hasta el commit o abort.
- Asegura **SR**. Garantiza **RC**, **ACA** y **ST** pero no es libre de deadlock.

## 2PL Rigoroso (2PLR)

- Es **2PL** y **todos los locks** (lectura y escritura) se mantienen hasta el final de la transacción.
- Asegura **SR**. Garantiza **RC**, **ACA** y **ST** pero no es libre de deadlock. La serializabilidad es igual al orden de los commit.

## Deadlocks

Ocurre cuando un conjunto de transacciones se bloquean mutuamente esperando recursos. Ninguna transacción puede continuar porque todas esperan que otra libere un lock.

### Detección: Wait-For Graph (WFG)

- Nodo por transacción.
- Arco  $T_i \rightarrow T_j$  si  $T_i$  espera un recurso que  $T_j$  mantiene.
- Si el grafo tiene ciclos, hay deadlock.
- Se selecciona una víctima para abortar y romper el ciclo:
  - **Antigüedad de la transacción:** Es preferible abortar una transacción más joven que una que lleva más tiempo ejecutándose.
  - **Cantidad de datos modificados:** Es mejor abortar la transacción que haya realizado menos modificaciones (menos registros en el log).
  - **Trabajo restante:** Si es posible estimarlo, conviene abortar la transacción con más operaciones pendientes.
  - **Número de ciclos en los que participa:** Mientras más ciclos de espera involucre, más conveniente abortarla.

## Prevención de Deadlocks

**Basada en Timestamps:** Si  $T_i$  intenta realizar un lock sobre un ítem y no puede porque  $T_j$  ya tiene un lock previo entonces hay dos estrategias:

- **Wait-Die:**
  - $TS(T_i) < TS(T_j)$ :  $T_i$  espera.
  - $TS(T_i) > TS(T_j)$ :  $T_i$  aborta.

- **Wound-Wait:**

- $TS(T_i) < TS(T_j)$ :  $T_j$  aborta.
- $TS(T_i) > TS(T_j)$ :  $T_i$  espera.

## Evitación de Deadlocks

- **Timeout:** si una transacción espera demasiado, se aborta.
- **No-Waiting:** si no se puede obtener el lock de inmediato, se aborta directamente.
- **Cautious Waiting:** se espera sólo si el lock está mantenido por una transacción que no espera a otra (para evitar ciclos).

## Concurrencia Optimista

El **enfoque optimista** asume que los conflictos entre transacciones son poco frecuentes, por lo tanto, permite ejecutar las operaciones **sin bloqueos** y valida la consistencia antes del commit.

### Comportamiento Físicamente Irrealizables

- **R2L (Read Too Late):**  $TS(T) < WT(x)$ . La transacción intenta leer un valor que fue escrito por una transacción posterior. No se puede garantizar consistencia temporal.
- **W2L (Write Too Late):**  $WT(x) < TS(T) < RT(x)$ . La transacción intenta escribir un valor cuando ya fue leído por otra transacción más nueva.

### Lectura sucia

- Ocurre cuando una transacción  $T$  lee un valor que fue escrito por otra transacción que aún no ha hecho commit ( $c(x) = F$ ).
- En concurrencia optimista, este caso se controla mediante espera hasta que  $c(x) = T$ .

### Thomas Write Rule (TWR)

- Evita abortar transacciones cuando una escritura llega tarde,  $TS(T) < WT(x)$ .
- Se ignora la escritura si ya hay un valor más nuevo confirmado y no hay lecturas pendientes.
- Aumenta el grado de concurrencia y reduce abortos innecesarios.



## Control por Timestamp

- Se basa en el uso de **timestamps** para ordenar operaciones y evitar conflictos.
- Cada transacción  $T$  tiene un **timestamp**  $TS(T)$  asignado al iniciarse.
- Las operaciones se autorizan o rechazan según los valores de **lecturas previas** y **escrituras previas** sobre el dato.

El scheduler mantiene, para cada ítem de datos  $x$ :

- **RT(x)**: el mayor timestamp de una transacción que **leyó**  $x$ .
- **WT(x)**: el mayor timestamp de una transacción que **escribió**  $x$ .
- **c(x)**: indica si el último valor de  $x$  fue **confirmado (T)** o aún no **commiteado (F)**.

## Planificador (Scheduler)

**Al recibir una lectura**  $r_T(x)$ :

- Si  $TS(T) \geq WT(x)$ :
  - Si  $c(x) = T$ , se concede la lectura. Se actualiza  $RT(x) := \max(RT(x), TS(T))$ .
  - Si  $c(x) = F$ , se demora hasta que el valor sea confirmado.
- Si  $TS(T) < WT(x)$ : **rollback** de  $T$  por conflicto de tipo **Read Too Late (R2L)**.

**Al recibir una escritura**  $w_T(x)$ :

- Si  $TS(T) \geq RT(x)$  y  $TS(T) \geq WT(x)$ :
  - Se permite la escritura. Se actualiza  $WT(x) := TS(T)$  y  $c(x) := F$ .
- Si  $TS(T) \geq RT(x)$  y  $TS(T) < WT(x)$ :
  - Si  $c(x) = T$ : se aplica la **Thomas Write Rule** y se ignora la escritura.
  - Si  $c(x) = F$ : se demora hasta que se confirme el valor.
- Si  $TS(T) < RT(x)$ : **rollback** de  $T$  por conflicto de tipo **Write Too Late (W2L)**.

## Control por Multiversión

- Cada transacción accede a las versiones de los datos vigentes al momento de su inicio.
- Se guardan tantas versiones como sean necesarias para atender a la transacción más vieja activa.
- No se conservan las versiones generadas por transacciones abortadas.

- Evita Read Too Late (R2L) y **NO** Write Too Late (W2L). El único tipo relevante de conflictos son los pares de operaciones **read-write** en el mismo ítem de datos.
- Grafo de conflictos:
  - Nodo por transacción.
  - Eje  $T_i \rightarrow T_j$  si  $T_j$  lee una versión escrita por  $T_i$  ( $r_j(x_i) \in H$ ).
  - Si el grafo es acíclico, la historia es serializable multiversión.

**Lectura:** se accede a la última versión escrita por una transacción no abortada con timestamp menor o igual al de la transacción lectora.

**Escritura:** se crea una nueva versión con el timestamp de la transacción escritora.

**Formalización:**

- $h(w_i(x)) = w_i(x_i)$ : crea una versión  $x_i$
- $h(r_i(x)) = r_i(x_j)$ : donde  $x_j$  fue la última versión válida antes del timestamp de  $T_i$
- $h(c_i) = c_i$ ,  $h(a_i) = a_i$ : los commits y abortos no cambian su forma

**Rechazo de operaciones:** se rechaza una operación cuando viola la consistencia de versiones. El único tipo relevante de conflictos son los pares de operaciones read-write en el mismo ítem de datos (W2L).

**Ventaja:** elimina bloqueos y permite gran concurrencia sin interferencia entre lecturas y escrituras.

**Desventaja:** mayor uso de espacio y necesidad de recolección de versiones antiguas.

## Multiversion Time Ordering (MVTO)

- **Lectura**  $r_i(x)$ : accede a  $x_k$  con el mayor  $TS(T_k) \leq TS(T_i)$ .
- **Escritura**  $w_i(x)$ :
  - Si existe  $r_j(x_k)$  tal que  $TS(T_k) < TS(T_i) < TS(T_j)$ : hay W2L y se aborta  $T_i$ .
  - Si no, se genera  $w_i(x_i)$  como nueva versión y es ejecutada.
- **Commit**  $c_i$ : se retrasa hasta que los commit  $c_j$  de todas las transacciones  $t_j$  que han escrito **nuevas versiones** de los elementos de datos leídos por  $t_i$  hayan sido ejecutados.

## Multiversion 2PL (MV2PL)

El MV2PL es un protocolo basado el locking usando 2PL riguroso.

- Se distinguen:
  - Versiones **commiteadas** (estables).
  - Versiones **no commiteadas** (de transacciones activas).

- **Lectura:** accede a la versión commiteada más reciente de  $x$ . Puede acceder a versión no commiteada si no hay conflicto.
- **Escritura:** se realiza cuando se libera el lock de escritura o la transacción commitea.
- **Commit:**  $T_i$  espera a que hayan commiteado:
  - Las transacciones que hayan leído la versión actual de un elemento de datos escrito por  $T_i$ .
  - Las transacciones de las que  $T_i$  leyó datos.

## 2V2PL – Two-Version Two-Phase Locking

- Variante de MV2PL que mantiene a lo sumo dos versiones por ítem:
  - Una versión **commiteada**.
  - Una versión **no commiteada**.
- **Lectura:** solo accede a la última versión commiteada.
- **Escritura:** se ejecuta al liberar el lock de escritura, (commit de la versión no commiteada que pasa a ser la versión commiteada), generando nueva versión no commiteada.
- **Commit:**  $T_i$  solo puede commitear cuando hayan commiteado:
  - Todas las transacciones  $T_j$  que leyeron la versión actual de un elemento de datos escrito por  $T_i$
  - Todas las transacciones  $T_j$  de las que  $T_i$  leyó datos

Este protocolo emplea tres tipos de **locks** para manejar versiones y garantizar serializabilidad:

- **rl (Read Lock):** Se adquiere antes de cada operación de lectura  $r(x)$  sobre la versión actual de  $x$ . Garantiza que la lectura se haga sobre una versión válida sin interferencias.
- **wl (Write Lock):** Se adquiere antes de cada operación de escritura  $w(x)$  para crear una versión **no commiteada** de  $x$ . Su función principal es asegurar que solo exista una versión no confirmada por elemento de datos en cualquier momento.
- **cl (Commit o Certify Lock):** Se adquiere justo antes del commit sobre cada elemento  $x$  que la transacción ha modificado. En este punto, los **write locks (wl)** se convierten en **certify locks (cl)**.

|         | $rl(x)$ | $wl(x)$ | $cl(x)$ |
|---------|---------|---------|---------|
| $rl(x)$ | +       | +       | —       |
| $wl(x)$ | +       | —       | —       |
| $cl(x)$ | —       | —       | —       |

Figura 15: Protocolo 2PL

## Historia de Operaciones y Locks

Historia original:

$r_1(x); w_2(y); r_1(y); w_1(x); c_1; r_3(y); r_3(z); w_3(z); w_2(x); c_2; w_4(z); c_4; c_3.$

Secuencia con locks:

$rl_1(x); r_1(x_0); wl_2(y); w_2(y_2); rl_1(y); r_1(y_0); wl_1(x); w_1(x_1); cl_1(x); ul_1; c_1; rl_3(y); r_3(y_0); rl_3(z);$   
 $r_3(z_0); wl_3(z); w_3(z_3); wl_2(x); w_2(x_2); cl_2(x); cl_3(z); ul_3; c_3; cl_2(y); ul_2; c_2; w_4(z_4); cl_4(z); ul_4; c_4.$

- Los **certify locks** cumplen el mismo rol que los **write locks** en el esquema de locking monoversión: garantizan que las versiones confirmadas de los datos respeten la serializabilidad multiversión.
- La compatibilidad entre **read locks (rl)** y **certify locks (cl)** define los ordenamientos válidos para:
  - Lecturas de versiones actuales.
  - Creación de nuevas versiones.
- A diferencia de 2PL monoversión, los **certify locks** se adquieren solo en la fase final de la transacción y se mantienen por un tiempo mucho más corto, lo que:
  - Reduce significativamente la contención entre transacciones.
  - Ofrece una **ventaja de rendimiento** frente a 2PL tradicional, sin sacrificar consistencia.

## Read Only Multiversión (ROMV)

ROMV es un protocolo híbrido que busca combinar la simplicidad del **(Strict) Two-Phase Locking (S2PL)** o del **Timestamp simple** con los beneficios de rendimiento de los esquemas multiversión.

- Las **transacciones de solo lectura** deben ser marcadas como tales al inicio.
- Las **transacciones de actualización** siguen el protocolo convencional **2PL**. A diferencia de un sistema monoversión, cada operación de escritura crea una nueva versión del dato en lugar de sobrescribirlo. Cada versión se asocia con el **timestamp de la transacción**, correspondiente al momento de su commit.
- Las transacciones de solo lectura utilizan un enfoque similar al **MVTO (Multiversion Timestamp Ordering)**:
  - Su timestamp es el del momento en que comienza la transacción.
  - Siempre acceden a la versión del dato con el **timestamp más alto** que sea **menor que el timestamp de la transacción**.

## Codificación, texto y tiempo

### Textos y codificación

- Una **codepage** define los caracteres y sus códigos numéricos.
- **ANSI** usa 1 byte por carácter (limitado a 256 símbolos).
- **Unicode** soporta caracteres de múltiples alfabetos y símbolos.
- Principales formatos: **UTF-8** (1–4 bytes), **UTF-16** (2–4 bytes) y **UTF-32** (4 bytes).

### Collation (Ordenamiento)

Define cómo comparar y ordenar texto en bases de datos:

- Afecta orden, acentos y mayúsculas.
- Impacta en claves y consultas (**WHERE**).
- Puede variar según idioma (ej.: tratamiento de ñ).

### Fechas y horas

- Existen múltiples calendarios históricos y zonas horarias.
- Se usa el estándar **ISO-8601** para formato de fechas y horas.
- Las bases de datos usan **timestamp** (fecha y hora juntas).
- Debe considerarse horario de verano, sincronización de servidores y segundos intercalares.

### Medición de tiempo

- Restar fechas produce un **intervalo**, no una fecha.
- PostgreSQL maneja intervalos con el tipo **INTERVAL**.
- ISO-8601 también define intervalos como texto (ej.: **P1Y2M10DT2H30M**).

## 7. Recovery Manager y Backups

### Data Base Management System(DBMS)

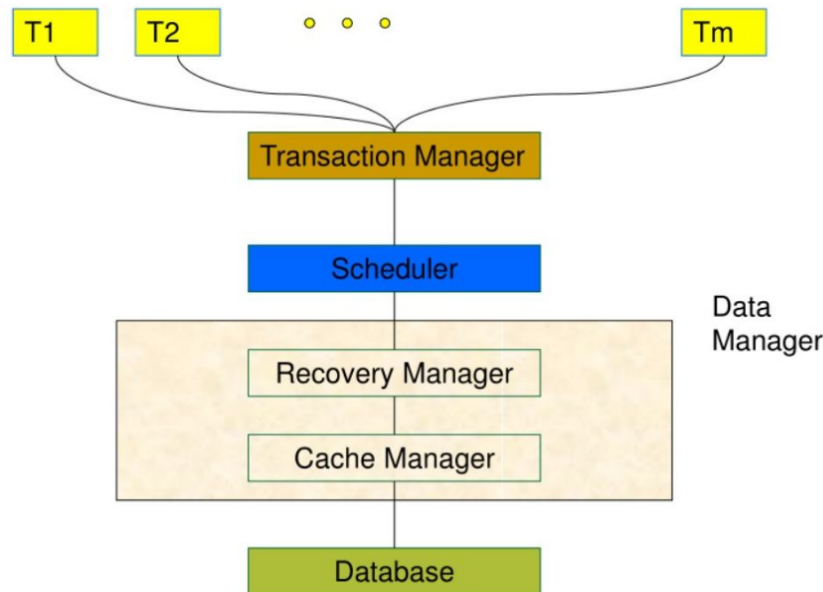


Figura 16: Componentes DBSM.

#### Componentes DBSM

- **Transaction Manager:** controla el inicio, *commit* y *abort* de las transacciones.
- **Scheduler:** coordina el orden en que se aplican las operaciones para garantizar consistencia, usualmente mediante técnicas como serializabilidad.
- **Data Manager:** se encarga del acceso eficiente y seguro a los datos almacenados. Incluye:
  - **Cache Manager:** administra la caché de páginas en memoria, incluyendo el manejo de *dirty bits*, operaciones de *fetch* (lectura desde disco o memoria) y *flush* (escritura a disco).
  - **Recovery Manager:** mantiene la durabilidad y recupera el estado consistente de la base de datos ante fallas del sistema.

El DBMS debe procurar que, al recibir una transacción, todas sus operaciones sean ejecutadas exitosamente, almacenando sus resultados en la base de datos (**commit**), o que ninguna operación tenga efecto sobre ella (**abort**).

#### Tipos de fallas

- **Fallo de la computadora (system crash):** debido a errores de hardware, software o de red.

- **Falla en la transacción o en el sistema:** errores en la lógica del programa (por ejemplo, división por cero) o interrupciones forzadas por el usuario.
- **Errores locales o condiciones de excepción:** por ejemplo, cuando no se encuentran los datos necesarios para completar una operación.
- **Ejecución del control de concurrencia:** abortos debidos a violaciones de serialización o a la detección de *deadlocks*. Estas transacciones pueden ser reintentadas posteriormente.
- **Falla de disco:** errores durante la lectura o escritura en disco.
- **Problemas físicos o catástrofes:** incluyen fallas en el suministro eléctrico, mal funcionamiento del aire acondicionado, incendios, robos, sabotajes, entre otros.

Salvo en los dos últimos casos (falla de disco y catástrofes), el sistema deberá poder mantener suficiente información para reestablecerse rápidamente en un modo a prueba de fallas. Si ocurre una falla, el DBMS debe:

- En caso de **commit**, asegurar que los cambios persistan en disco.
- En caso de **abort**, garantizar que los efectos parciales se deshagan y que la base de datos recupere un estado consistente.
- **Rollback:** es la operación que revierte todos los cambios realizados por una transacción, devolviendo los datos al estado anterior al inicio de dicha transacción. Ocurre siempre que una transacción termina en *abort*.

## El Log de Transacciones

El **log** registra cada operación crítica para la BD que realiza una transacción de manera de reestablecer el sistema ante fallas. Es **incremental y secuencial**, se guarda en disco pero no se salva de fallas a disco ni catástrofes, por lo que periódicamente es preferente guardar una copia de seguridad en otro dispositivo. Su presencia hace que para que una transacción se considere comiteada sus cambios debieron haber sido registrados en el log.

### Reglas de Escritura (Write-Ahead Logging)

- Toda modificación debe registrarse primero en el log antes de aplicarse a disco.

### Garbage Collection Rule

Un registro  $[T_i, x, V]$  puede eliminarse del log si:

- $T_i$  fue abortada.
- $T_i$  comiteó, y hay una transacción  $T_j$  posterior que también comiteó y actualizó el mismo ítem:  $[T_j, x, W]$ .

## Estructura de los registros del Log

- <START  $T_i$ >: marca el inicio de la transacción  $T_i$ .
- <COMMIT  $T_i$ >: indica que la transacción  $T_i$  terminó exitosamente y sus efectos deben ser permanentes.
- <ABORT  $T_i$ >: indica que la transacción  $T_i$  fue abortada y sus efectos deben deshacerse.
- **Update record**: varía según el tipo de logging:
  - UNDO: < $T_i$ ,  $X$ ,  $v$ > donde  $v$  es el valor antiguo.
  - REDO: < $T_i$ ,  $X$ ,  $v$ > donde  $v$  es el nuevo valor.
  - UNDO/REDO: < $T_i$ ,  $X$ ,  $v_0$ ,  $v_1$ > donde  $X$  cambia de  $v_0$  a  $v_1$ .

Una transacción se considera incompleta si no contiene ni <COMMIT  $T_i$ > ni <ABORT  $T_i$ >.

## Procesos de Recovery

### UNDO Logging

- Se basa **deshacer** las **transacciones incompletas** que escribieron a disco.
- Registro: < $T_i$ ,  $X$ ,  $v$ > (valor anterior).
- El <COMMIT  $T_i$ > se escribe solo **después** de escribir los cambios en disco.
- **Recuperación**:
  1. Se recorre el log desde el final.
  2. Se identifican las transacciones incompletas.
  3. Por cada < $T_i$ ,  $X$ ,  $v$ > de transacciones incompletas: se restaura  $X = v$  en disco.
  4. Se escribe <ABORT  $T_i$ > incompleta en el log.
- Se pueden ir bajando datos a la DB durante la escritura del log para ahorrar tiempo.
- Requiere leer todo el archivo de log y que los ítems se bajen a disco justo después de que la transacción termine, aumentando la cantidad de operaciones de entrada y salida.

### Logging: REDO Logging

- Se basa **rehacer** las transacciones **completas** cuyos cambios no llegaron a bajarse a disco.
- Registro: < $T_i$ ,  $X$ ,  $v$ > (valor nuevo).
- El <COMMIT  $T_i$ > se escribe **antes** de escribir los cambios en disco.



- **Recuperación:**
  1. Se recorre el log desde el inicio.
  2. Se identifican transacciones completas.
  3. Por cada  $\langle T_i, X, v \rangle$  de transacciones completas: se aplica  $X = v$  en disco.
  4. Se escribe  $\langle \text{ABORT } T_i \rangle$  para cada  $T_i$  incompleta.
- En la recuperacion el archivo de log debe leerse dos veces y como las transacciones suelen ser completas va a haber muchos cambios a rehacer.
- Los cambios no pueden bajarse a la BD hasta luego del commit es necesario mantener los bloques modificados en un buffer.

### Logging: UNDO/REDO Logging

- Registro:  $\langle T_i, X, v_0, v_1 \rangle$  (cambio de  $v_0$  a  $v_1$ ).
- El  $\langle \text{COMMIT } T_i \rangle$  se escribe solo **después** de escribir los cambios en disco.
- **Recuperación:**
  1. **Primera pasada (hacia atrás): UNDO**
    - Identificar transacciones incompletas.
    - Restaurar  $X = v_0$ .
    - Escribir  $\langle \text{ABORT } T_i \rangle$ .
  2. **Segunda pasada (hacia adelante): REDO**
    - Para cada transacción completa: aplicar  $X = v_1$ .
- Esto hace que el sistema sea mas flexible a los escenarios de fallas, pero para ello debe guardar mas información y el trabajo de recuperación es mayor.

### Manejo de Multiverso en Recovery

- En sistemas con **MVCC**, las versiones generadas por transacciones abortadas se descartan o se marcan como inválidas.
- Solo se mantienen versiones activas y versiones escritas por transacciones que hayan hecho commit.
- Esto garantiza la consistencia y correcta visibilidad de los datos para otras transacciones concurrentes.

### Checkpoint

Los **checkpoints** son mecanismos utilizados por el *Recovery Manager* para reducir el tiempo y el espacio necesarios durante la recuperación, al marcar puntos seguros desde los cuales comenzar a procesar el log.

## Objetivo

- Limitar la porción del log que debe ser analizada durante una recuperación.
- Acelerar el proceso de `Restart()` en caso de fallos.
- Evitar el escaneo completo del log desde el inicio de los tiempos.

## Tipos de Checkpoints

- **Quiescente:**
  - El sistema **no permite iniciar nuevas transacciones** mientras se realiza el checkpoint.
  - Se espera que todas las transacciones activas finalicen.
  - Luego se escribe `<CKPT>` en el log y se fuerza a disco.
  - Se reanudan las transacciones nuevas.
  - Simple de implementar, pero implica una pausa temporal en el sistema.
- **No quiescente:**
  - **Permite nuevas transacciones** durante la ejecución del checkpoint.
  - Se registra el conjunto de transacciones activas al inicio: `<START CKPT( $T_1, \dots, T_k$ )>`.
  - No interrumpe el sistema, pero requiere mayor complejidad en la recuperación.

## Checkpoint Quiescente con UNDO Logging

- Se bloquea el ingreso de nuevas transacciones.
- Se espera a que todas las activas terminen.
- Se escribe `<CKPT>` en el log y se fuerza a disco.
- **En recuperación**
  - Aplicar UNDO desde el final del log hasta el último checkpoint (el primero encontrado)

## Checkpoint No Quiescente con UNDO Logging

- Se registra `<START CKPT( $T_1, \dots, T_k$ )>` en el log y flush del mismo.
- Esperar a que **todas las transacciones  $T_1, \dots, T_k$  comitéen o aborten**, pero sin restringir que empiecen nuevas transacciones
- Al finalizar, se escribe `<END CKPT>` y flush del mismo.
- **En recuperación**

- Aplicar UNDO desde el final del log hasta:
  - Si se encuentra un  $\langle \text{END CKPT} \rangle$ , seguir hasta el  $\langle \text{STARTCKPT}(T_1, \dots, T_k) \rangle$
  - Si se encuentra un  $\langle \text{START CKPT}(T_1, \dots, T_k) \rangle$ , seguir hasta el  $\langle \text{START } T_i \rangle$  más antiguo,  $1 \leq i \leq k$ . O sea, las que quedaron incompletas.

### Checkpoint No Quiescente con REDO Logging

- Se escribe  $\langle \text{START CKPT}(T_1, \dots, T_k) \rangle$  y flush del mismo.
- Forzar a disco los ítems que fueron escritos en el pool de buffers por **transacción que hicieron commit** al momento del  $\langle \text{Start CKPT} \rangle$  pero que aun no fueron guardados en disco.
- Luego se escribe  $\langle \text{END CKPT} \rangle$  y flush del mismo.
- **En recuperación**
  - Identificar el último  $\langle \text{Start CKPT}(\dots) \rangle$  que tiene un  $\langle \text{End CKPT} \rangle$  posterior. Tener en cuenta que el  $\langle \text{End CKPT} \rangle$  indica que las transacciones que hicieron commit antes del  $\langle \text{Start CKPT} \rangle$  tienen sus cambios en disco.
  - Si se encuentra un  $\langle \text{Start CKPT} \rangle$  y no hay  $\langle \text{End CKPT} \rangle$  debido a un crash. Ubicar el  $\langle \text{End CKPT} \rangle$  **anterior** y su correspondiente  $\langle \text{START CKPT}(T_1, \dots, T_k) \rangle$  y rehacer todas las transacciones comiteadas que comenzaron a partir de ahí o están entre las  $T_i$ .

### Checkpoint No Quiescente con UNDO/REDO Logging

- Se registra  $\langle \text{START CKPT}(T_1, \dots, T_k) \rangle$  y flush.
- Escribir a disco **todos los dirty buffers** al momento del  $\langle \text{Start CKPT} \rangle$ .
- Al finalizar, se escribe  $\langle \text{END CKPT} \rangle$  y flush.
- En recuperación:
  - **Transacciones incompletas:** para deshacerlas se debe retroceder hasta el start más antiguo de ellas. Agregar registro  $\langle \text{ABORT } T_i \rangle$  al log para cada transacción incompleta, y hacer flush del log los cambios desde el  $\langle \text{START CKPT} \rangle$  correspondiente.
  - **Transacciones con commit:** Si leyendo desde el último registro se encuentra un  $\langle \text{END CKPT} \rangle$  sólo será necesario rehacer las acciones efectuadas desde el correspondiente registro  $\langle \text{START CKPT}(T_1, \dots, T_k) \rangle$  en adelante. Si no encontramos el  $\langle \text{END CKPT} \rangle$ , usamos la estrategia del REDO.

## Transacciones de larga duración

Casos típicos:

- Modificación masiva de registros.
- Workflows entre sistemas (ej: recarga SUBE desde home banking).

### Long duration: Transacciones masivas

- Inserciones masivas: usar `COPY FROM`.
- A veces la base de datos no puede ejecutarlo como una única transacción por el tiempo que tarda en ejecutarse y por la cantidad de información que necesita almacenar para poder hacer rollback
- Problemas: rollback costoso, triggers, índices.
- No existe una instrucción especial para `UPDATE` masivos.

### Long duration: Transacciones inter sistemas

Para implementar transacciones de larga duración que involucren múltiples sistemas independientes se utiliza el concepto de **saga distribuida**.

- Una **saga** es un conjunto de acciones que, juntas, conforman una *long duration transaction*.
- Dado que no se puede utilizar una única transacción atómica en múltiples sistemas, se implementan mecanismos de **compensación**, que revierten los efectos de acciones anteriores si alguna etapa falla.

### Estructura de una saga

- Se representa mediante un **grafo dirigido**, cuyos nodos son:
  - Las acciones individuales (A, B, C, ...)
  - El nodo **Abort**
  - El nodo **Complete**
- Cada camino en el grafo representa un posible **curso de ejecución**.
- Se define una marca que indica cuál es el nodo inicial.

## Compensación y atomicidad lógica

- Para garantizar una **atomicidad lógica**, cada acción se considera una transacción individual.
- A cada acción  $A$  le corresponde una **acción de compensación**  $A^{-1}$ , tal que:

$$A \circ A^{-1} \Rightarrow \text{estado original}$$

- Si la saga debe revertirse, se ejecutan las compensaciones en el **orden inverso** al de ejecución.

**Advertencia:** si una acción de compensación también falla, se entra en un escenario complejo que puede requerir intervención manual o estrategias avanzadas de recuperación.

## Ejemplo

En un sistema como la recarga de SUBE desde el home banking, cada paso (registro de transacción en Link, banco, y SUBE) se convierte en una acción dentro de la saga. Si uno de estos pasos falla, las operaciones previas deben deshacerse con acciones de compensación para evitar inconsistencias entre sistemas.

## SAGA DE LA SUBE

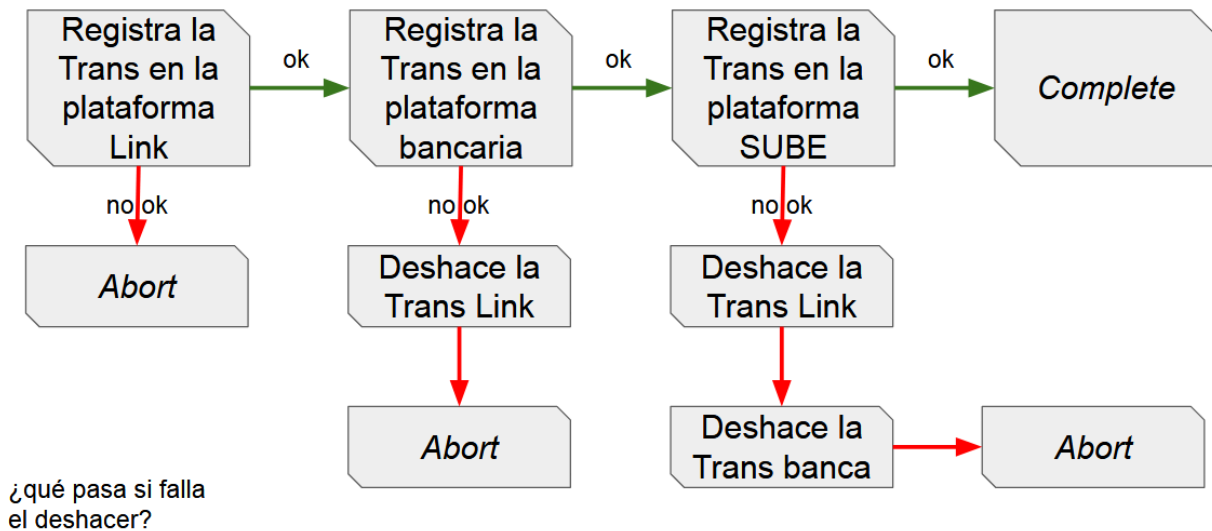


Figura 17: Saga de la SUBE

## Backups

### Objetivos:

- Recuperar datos perdidos.
- Migrar o auditar bases de datos.
- Trabajar sobre una copia para pruebas o desarrollo.

### Tipos de backup:

- **Muleto**: réplica sincronizada.
- **Hot backup**: se realiza sin detener el sistema.
- **Backup binario**: formato propietario, rápido y compacto.
- **Dump de texto**: conjunto de comandos SQL, útil para migraciones entre motores o versiones.

## Tipos de Backups

### Hot Backup

- También conocido como **backup en caliente**, permite hacer una copia sin detener la base de datos.
- Se utiliza un comando común del sistema operativo (por ejemplo, `tar.gz`).
- El procedimiento consiste en:
  - Primero copiar los archivos de datos.
  - Luego, copiar los archivos de log.
- Aunque la base siga en funcionamiento, los logs son **estrictamente secuenciales**, lo que permite una recuperación correcta.

### Ventajas:

- No requiere detener ni enlentecer el funcionamiento de la base de datos.
- Simplicidad operativa: usa herramientas comunes del sistema operativo.

### Desventajas:

- Dependiente de la versión específica del motor de base de datos.
- Puede estar condicionado a la marca o versión del sistema operativo.

## Backup Binario

- Utiliza un formato propietario del motor de base de datos.
- Permite mover la base entre distintos sistemas operativos o versiones del motor (siempre que no se cambie de *major version*).

### Ventajas:

- Muy eficiente: rápido y compacto tanto al generar como al restaurar el backup.
- Suele mantener buena compatibilidad entre versiones.

### Desventajas:

- Puede requerir detener o ralentizar el uso normal de la base durante su ejecución.

## Dump de Texto (Logical Backup)

- Consiste en una exportación completa de la base como un archivo de texto con comandos SQL.
- Incluye:
  - Datos,
  - Esquemas,
  - Relaciones,
  - Restricciones,
  - Triggers,
  - Procedimientos almacenados,
  - Configuraciones adicionales (por ejemplo: dominios).

### Ventajas:

- Portabilidad: permite migrar datos entre motores de diferentes marcas.
- Es el mecanismo habitual para realizar **upgrades** de versión mayor del motor.

### Desventajas:

- Es más lento que otras alternativas.
- Ocupa más espacio, incluso comprimido.
- Puede requerir procesamiento adicional si hay elementos no estandarizados.

## 8. Optimización Física de Consultas

### Organización Física de los Registros

Los registros en bases de datos pueden almacenarse de distintas formas. Se asume por simplicidad que los registros son de longitud fija, aunque en la práctica pueden tener longitud variable (campos NULL, VARCHAR, BLOB).

### Factor de Bloque (FBR)

**Longitud de registro (LR):** se calcula como la suma de las longitudes de cada campo. El **factor de bloqueo** mide cuántos registros caben en un bloque:

- $FBR = \lfloor \frac{B}{LR} \rfloor$  (redondeo por defecto)
- $BR = \lceil \frac{TR}{FBR} \rceil$  (número de bloques para almacenar una tabla de  $TR$  registros)

Mide cuantos registros entran en un bloque

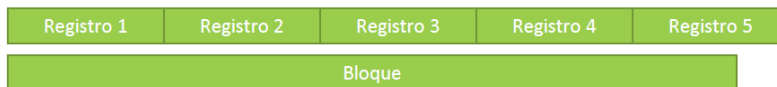


Figura 18: Ejemplo de cálculo del factor de bloque

### Estructura física de una Tabla y Particiones

Un archivo de base de datos se divide en páginas o bloques (unidad de I/O). Cada bloque puede contener:

- **Datos:** registros de tabla.
- **Lob Pages:** para datos grandes como CLOB o BLOB.
- **IAM Pages:** Information Allocation Map, para rastrear la asignación.
- **Directorio de páginas:** apuntadores a otras páginas.



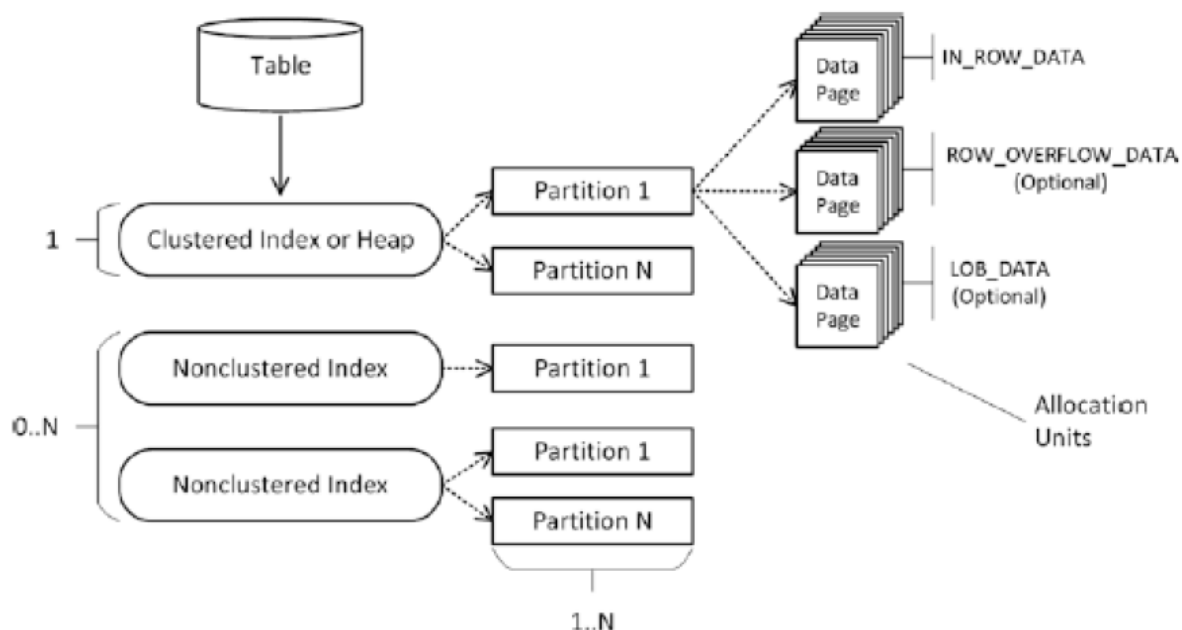


Figura 19: Estructura física de una tabla: particiones, índices y unidades de asignación

## Estructura de una Fila (Row)

En SQL Server, cada fila (row) dentro de una página de datos tiene una estructura interna compuesta por distintos campos que permiten gestionar la integridad, flexibilidad y eficiencia del almacenamiento.

La siguiente figura muestra la organización lógica de una fila, incluyendo tanto datos de longitud fija como variable, y campos de control.

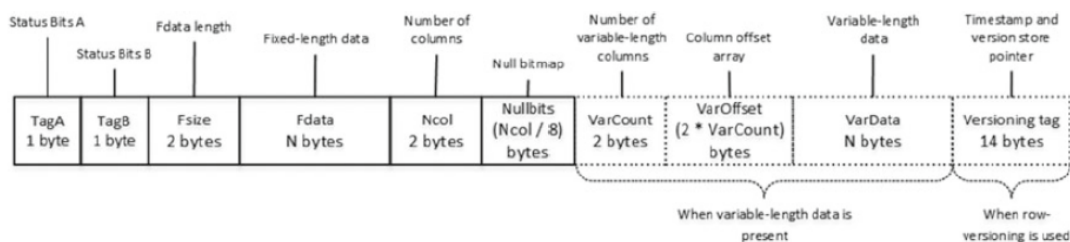


Figura 20: Estructura interna de una fila en SQL Server

Status Bits A y B contiene información sobre la fila: tipo de fila, si fue borrada lógicamente (ghosted), si tiene valores nulos, si tiene datos de longitud variable, etc.

## Datos LOB y estructura de almacenamiento

Los tipos de datos grandes como `VARBINARY(MAX)`, `VARCHAR(MAX)`, `TEXT` e `IMAGE` se almacenan en SQL Server mediante una estructura especial llamada **LOB root structure**.

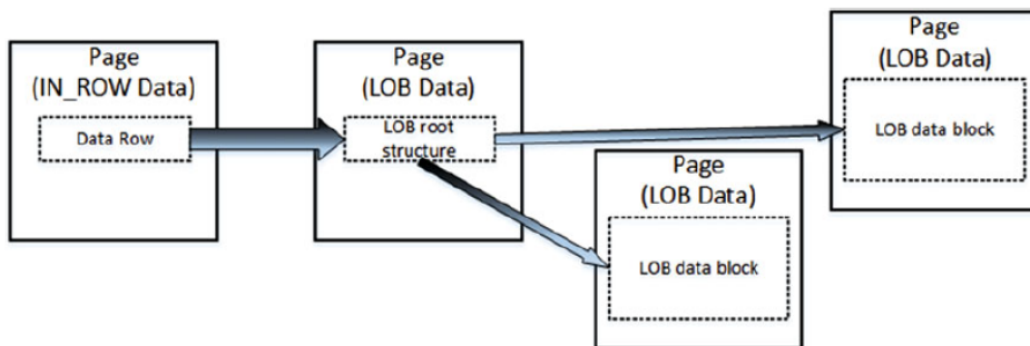


Figura 21: Estructura de almacenamiento de datos LOB. Una fila que contiene un dato LOB apunta a una LOB Root Structure, que a su vez referencia páginas con los fragmentos del objeto.

- **LOB Root Structure:** contiene punteros a las páginas donde se encuentran almacenados los fragmentos de los datos LOB.
- **Small LOB Data:** si el tamaño del dato LOB es pequeño (menor a 32 KB, y cabe en cinco páginas de datos), la LOB root structure puede hacer referencia directa a esas páginas.
- **Large LOB Data:** si el dato LOB es más grande, se utilizan niveles intermedios de páginas para formar una jerarquía de almacenamiento. Esta estructura es similar a un árbol B (B-Tree), con nodos intermedios que apuntan a nodos secundarios, y las hojas que contienen los datos reales.

## Páginas



Figura 22: Página en SQL Server: tipos y metadatos

En SQL Server, la unidad mínima de almacenamiento y de entrada/salida (I/O) es la **página**, con un tamaño fijo de **8 KB**.

Las páginas se agrupan de a ocho en estructuras llamadas **extents**. Cada operación de lectura o escritura a disco siempre involucra páginas completas.

### Buffer pool:

- SQL Server mantiene en memoria un **buffer pool** donde guarda las páginas accedidas recientemente.

- Si una página ya está en memoria, se hace una **lectura lógica**.
- Si no está en memoria, se realiza una **lectura física** desde disco seguida de una lectura lógica.
- Si una página en memoria fue modificada, se marca como **dirty**. Estas páginas se escriben a disco de forma diferida.

#### **Tipos de páginas comunes:**

- **File Header Page:** contiene metadatos del archivo.
- **Boot Page:** similar a la anterior, pero para toda la base de datos.
- **Page Free Space (PFS):** indica el estado de cada página (llena, vacía o parcialmente llena) y cuánto espacio libre tiene. Aparece en la página 2 del archivo y luego cada 8088 páginas.
- **IAM (Index Allocation Map):** mapea los extents usados por cada unidad de asignación (heap, índice, etc.). Cada página IAM cubre 64.000 extents ( 4 GB).
- **GAM (Global Allocation Map):** indica si un extent está libre (bit = 1) o asignado (bit = 0).
- **SGAM (Shared Global Allocation Map):** indica si un extent es mixto y tiene al menos una página libre.
- **DCM (Differential Changed Map):** señala qué extents cambiaron desde el último backup diferencial.
- **ML (Minimally Logged):** indica si un extent fue modificado por una operación con logging mínimo.
- **Páginas de datos:** contienen las filas de las tablas.
- **LOB Pages:** páginas especiales para almacenar datos grandes como VARCHAR(MAX) o BLOB.
- **Páginas de directorio o raíz:** usadas en estructuras como árboles B+ para apuntar a otras páginas.

#### **Páginas IAM (Index Allocation Map)**

Las **páginas IAM** (Index Allocation Map) son estructuras clave en SQL Server que permiten rastrear qué extents están asignados a una unidad de asignación específica dentro de una partición (ya sea un heap o un índice).

Cada página IAM es un **mapa de bits** donde cada bit representa un extent dentro de un intervalo. Si el bit está en 1, ese extent pertenece a la unidad de asignación correspondiente.

- Cada página IAM cubre aproximadamente **64.000 extents**, lo que equivale a casi **4 GB de datos**.

- Cuando una tabla o índice supera este tamaño, se crean y enlazan múltiples páginas IAM, formando una **cadena IAM**.
- Cada página IAM cubre un intervalo de extents definido por una página GAM (Global Allocation Map).
- Gracias a las IAM, SQL Server puede localizar rápidamente todas las páginas de datos asociadas a una tabla sin tener que recorrer todo el archivo.

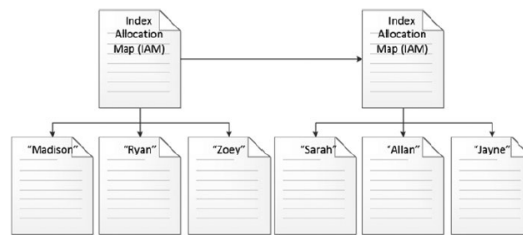


Figura 23: Heap: tabla sin índice clustered, con IAM que apunta a las páginas de datos

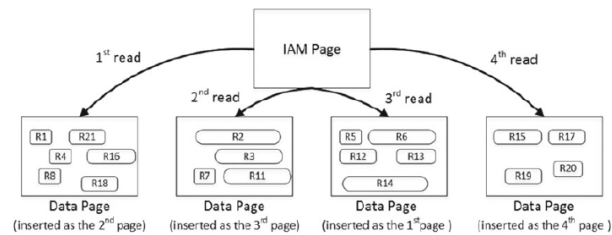


Figura 24: Mapa de Asignación de Índices (IAM)

## Heap Table

Una **heap table** es una tabla sin índice clustered. Los registros no tienen un orden físico específico: se insertan en la primera página disponible según la asignación del sistema.

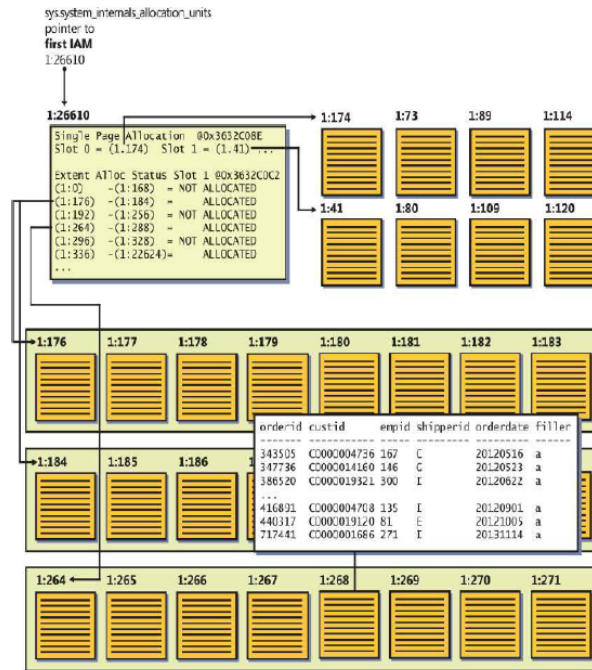


Figura 25: Heap

- SQL Server mantiene una o más **páginas IAM** (Index Allocation Map) que indican en qué extents se encuentran las páginas de datos del heap.
- Cuando se realiza una consulta sobre una tabla heap, SQL Server utiliza estas páginas IAM para identificar qué páginas deben escanearse.
- El escaneo de datos se hace en orden de asignación (no de inserción).

#### Actualización de registros:

- Si se actualiza una fila y hay espacio en la misma página, se reescribe allí mismo.
- Si no hay espacio, la nueva versión se mueve a otra página y la original es reemplazada por un **puntero de redireccionamiento** (fila de 16 bytes) que apunta a la nueva ubicación.
- A esta nueva versión se la llama **fila redireccionada**.

## Árboles B+

Estructura de índice balanceada, adecuada para lecturas secuenciales y búsquedas eficientes. Todos los datos están en las hojas y las hojas están enlazadas entre sí.

## ARBOLES B+

Son árboles B, balanceados

Nodo Terminal



Apunta al bloque que tiene los registros de clave  $K_i$

Apunta al próximo nodo terminal

Nodo Intermedio



Apunta al nodo con claves menores a  $K_1$

Apunta al nodo con claves  $\geq K_{i-1}$  y menores que  $K_i$

Apunta al nodo con claves  $\geq K_{d-1}$

Figura 26: Árbol B+: nodos internos con claves, hojas con punteros a datos

Los índices en SQL Server se implementan mediante **árboles B+**, una estructura balanceada de múltiples niveles que garantiza tiempos de búsqueda eficientes ( $O(\log n)$ ) incluso con grandes volúmenes de datos.

- Un árbol B+ está compuesto por:
  - **Nivel raíz:** punto de entrada para cualquier búsqueda.
  - **Niveles intermedios** (si existen): contienen claves mínimas y punteros a nodos inferiores.
  - **Nivel hoja:** contiene todas las claves y punteros a los datos reales (o RIDs).
- Las hojas están **enlazadas secuencialmente**, lo cual permite:
  - Recorridos ordenados eficientes.
  - Ejecución rápida de consultas con condiciones de rango ('BETWEEN', ' $\geq$ ', etc.).
- Todos los datos están en el nivel hoja (a diferencia de un árbol B clásico).
- El árbol se reequilibra automáticamente ante inserciones o eliminaciones, manteniendo el orden y la profundidad mínima.

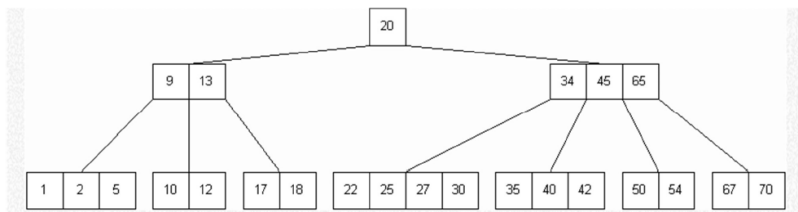


Figura 27: Ejemplo de árbol B+ completo con enlaces entre hojas

## Índices

Un **índice** es una estructura de datos que funciona como un diccionario de claves (que pueden ser únicas o no) asociadas a una relación. Cada clave corresponde a una o más columnas (atributos) de la tabla, y su valor asociado puede ser el registro completo (tupla), su row identifier (RID) o una lista de RIDs de todos los registros que comparten esa clave.

Los índices pueden ser **densos** o **no densos**, según si contienen una entrada para cada registro de la tabla (densos) o solo para algunos registros (no densos).

Además, se clasifican en:

- **Índices primarios:** almacenan los registros completos del archivo (solo puede existir uno por tabla). No admiten duplicados y contienen los datos junto al propio índice.
- **Índices secundarios:** son todos aquellos que no son primarios. El archivo de datos está separado del índice, y en lugar de registros completos, almacenan RIDs. Puede haber varios por tabla y pueden contener claves duplicadas.

## Índice Clustered

Los datos están físicamente ordenados por la clave primaria. Un bloque contiene registros ordenados y el índice apunta a bloques.

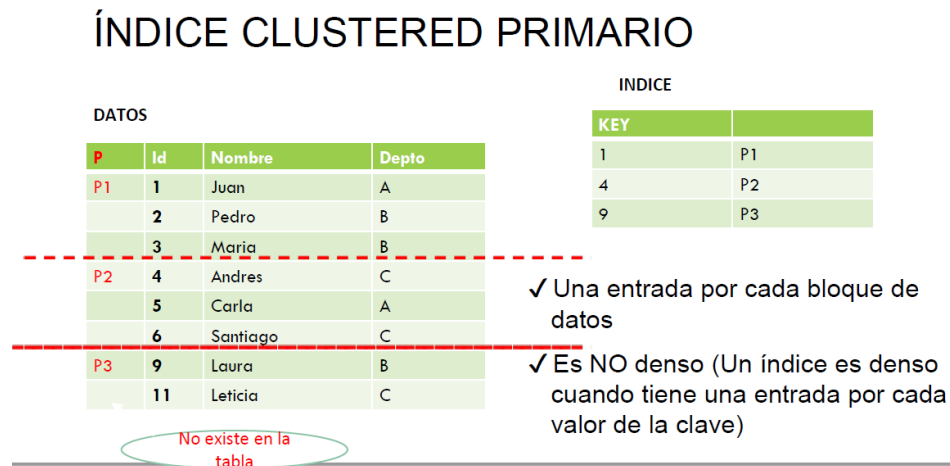


Figura 28: Índice clustered primario: apunta a bloques de datos ordenados

- Un **índice clustered** define el **orden físico** de los datos en la tabla.
- Solo puede haber un índice clustered por tabla.
- Siempre hay un **nivel hoja**, cero o más niveles intermedios y un **nivel raíz**.
- Si todos los datos caben en una única página, entonces el árbol B se reduce a esa única página.
- En los niveles intermedios se almacena:

- Dirección física de la página hija.
- Valor mínimo de la clave contenida en esa página hija.
- (Excepción) En la primera fila de la primera página se almacena NULL para optimizar inserciones al inicio.

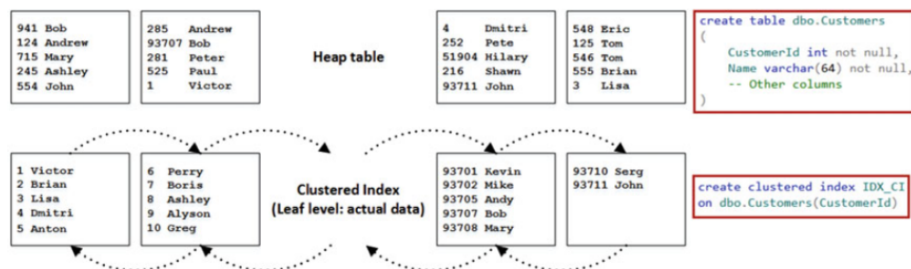


Figura 29: Comparacion Heap vs Clustered Index

## Índice Non-Clustered

Los datos no están ordenados por la clave del índice. El índice es denso y contiene un puntero por cada valor distinto de la clave.

### ÍNDICE NON CLUSTERED

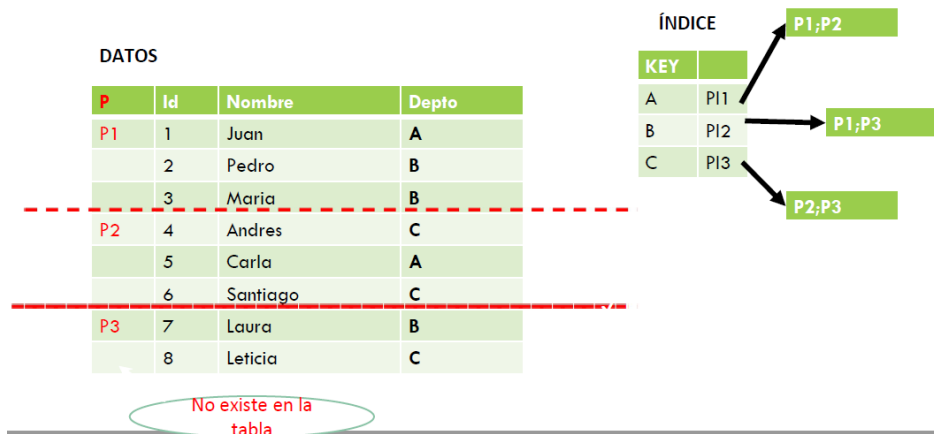


Figura 30: Índice non-clustered: entradas por cada valor de la clave

- Los índices non-clustered definen un orden lógico diferente al de los datos y se almacenan en una estructura separada.
- Estructuralmente son similares a los clustered: árbol B+ con nodos intermedios y hojas.
- En las hojas, cada entrada incluye:



- El valor de la clave.
- Un **row ID (RID)** que apunta al registro real:
  - En tablas heap: el RID indica la página física.
  - En tablas con índice clustered: el RID es la clave del índice clustered.
- Los nodos intermedios contienen punteros físicos y claves mínimas para dirigir la búsqueda.

## Índice Hash

Se implementa como una **tabla de hash** con un número fijo de *buckets*. Es especialmente eficiente para **búsquedas por igualdad**, ya que su costo de acceso es proporcional a la cantidad máxima de bloques por bucket ( $MB \times BI$ ). Sin embargo, para otros tipos de consultas (como rangos), se vuelve ineficiente, ya que requiere realizar un barrido lineal completo del archivo (*file scan*).

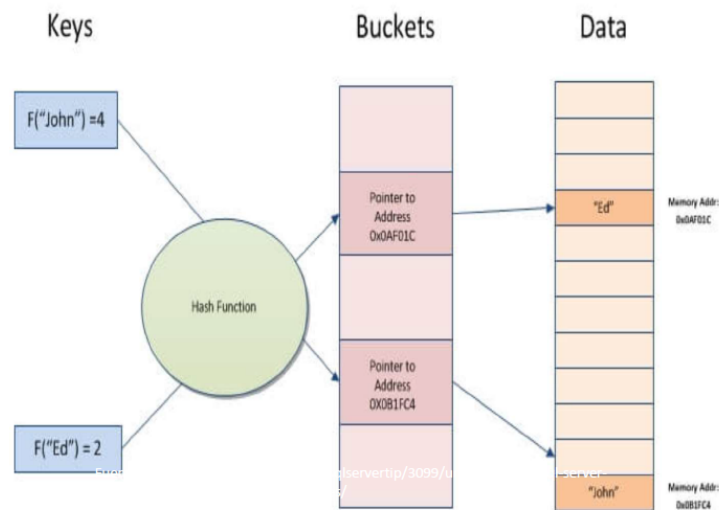


Figura 31: Índice hash: acceso directo a buckets

- La clave es procesada por una **función hash** que devuelve un número de bucket.
- Cada bucket contiene un puntero al registro correspondiente o a una lista de registros con claves que colisionaron.
- Este tipo de índice es muy rápido para consultas de la forma 'WHERE A = valor'.
- No se puede usar eficientemente para búsquedas por rangos ('<', '>', 'BETWEEN') ni para recorridos ordenados.
- En SQL Server, los índices hash se usan exclusivamente en **tablas optimizadas para memoria** ('memory-optimized tables').

## Índices Compuestos

Un **índice compuesto** es aquel que incluye más de una columna en su definición.

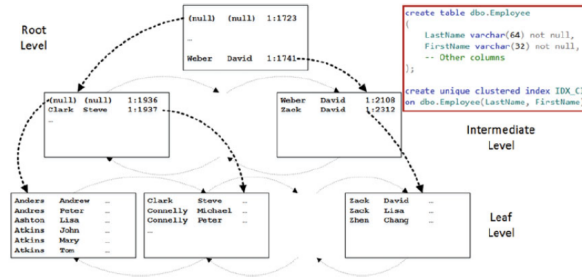


Figura 32: Índice Compuesto

- El orden de las columnas en el índice es crucial: afecta tanto el uso para filtrado como para ordenamiento.
- Se puede aprovechar un índice compuesto para consultas que:
  - Filtran por la primera columna o por un prefijo de columnas.
  - Ordenan por un subconjunto de las columnas del índice.
- La **SARGabilidad** (Search Argument-able) depende de que las condiciones del 'WHERE' permitan uso eficiente del índice (sin funciones sobre columnas, por ejemplo).

## Índice Multinivel

Cuando el índice es muy grande, se puede construir un segundo nivel de índice sobre el primero. Esto mejora el acceso.

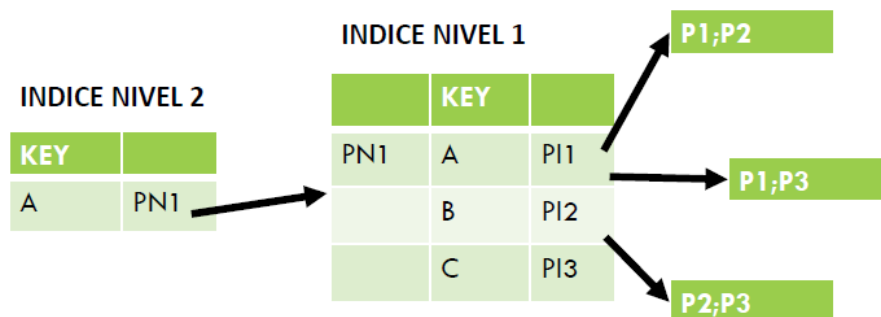


Figura 33: Índice multinivel: niveles de accesos intermedios

## Costo de Búsqueda

Sea  $A$  un atributo,  $a$  un valor buscado, y:

- $B_R$ : número de bloques de datos.
- $T'$ : número de tuplas que cumplen la condición.
- $FB_R$ : número de tuplas por bloque.
- $X$ : altura del índice B+.
- $MB$ : número de páginas del índice hash.
- $B$ : profundidad del árbol hash.

| TIPO ARCHIVO             | $A = a$                               | $A > a$ |
|--------------------------|---------------------------------------|---------|
| SIN INDICE ( HEAP FILE ) | $B_R$                                 | $=$     |
| SORTED FILE              | $\log_2(B_R) + \lceil T'/FB_R \rceil$ | $=$     |
| INDICE B+ CLUSTERED      | $X + \lceil T'/FB_R \rceil$           | $=$     |
| INDICE B+ UNCLUSTERED    | $X + T'$                              | $=$     |
| HASH INDEX               | $MB \times B + T'$                    | $B_r$   |

- $A$ : atributo por el cual se busca.
- $a$ : valor específico del atributo.
- $B_R$ : número de bloques en el archivo de datos.
- $T'$ : número de tuplas que cumplen la condición.
- $FB_R$ : número de tuplas por bloque.
- $X$ : altura del índice B+.
- $MB$ : número de páginas del índice hash.
- $B$ : profundidad del árbol hash.

Bases de Datos - D

Figura 34: Tabla

### 1. Sin Índice (Heap File)

- Los datos no tienen orden ni índice.
- Para  $A = a$  (o  $A > a$ ), se debe leer toda la relación:

$$Costo = B_R$$

## 2. Archivo Ordenado (Sorted File)

- Los datos están ordenados físicamente por  $A$ .
- Para  $A = a$ :

$$Costo = \log_2(B_R) + \left\lceil \frac{T'}{FB_R} \right\rceil$$

(búsqueda binaria + lectura de bloques relevantes).

- Para  $A > a$ , se leen secuencialmente las tuplas desde la posición inicial hasta el final.

## 3. Índice B+ Clustered

- El índice B+ coincide con el orden físico de los datos.
- Para  $A = a$ :

$$Costo = X + \left\lceil \frac{T'}{FBR} \right\rceil$$

(descenso por el árbol + lectura secuencial de bloques).

- Para  $A > a$ , el acceso secuencial es eficiente gracias al orden físico.

## 4. Índice B+ Unclustered

- El índice B+ no coincide con el orden físico de los datos.
- Para  $A = a$ :

$$Costo = X + T'$$

(cada tupla puede implicar un acceso a un bloque diferente).

- Para  $A > a$ , el costo crece mucho debido a accesos dispersos.

## 5. Índice Hash

- Utiliza una función hash para ubicar la clave.
- Para  $A = a$ :

$$Costo = MB \times B + T'$$

Utiliza una función hash para ubicar directamente el bucket donde se encuentra la clave buscada. Cada búsqueda implica:

- Acceder a las  $MB$  páginas del índice hash hasta llegar al bucket correspondiente.
- Recorrer una estructura con profundidad  $B$  (árbol o directorio del hash extendido) para localizar la página exacta.
- Leer las  $T'$  tuplas que cumplen la condición desde ese bucket.

- Para  $A > a$ :

$$Costo = B_R$$

(debe leerse toda la relación, ya que el hash no soporta rangos).

## Cálculo de $T'$

$T'$  representa el número estimado de registros que cumplen una condición en una tabla. Se asume una **distribución uniforme** de los valores de los atributos, es decir, que todas las claves y rangos tienen la misma probabilidad.

- Para una condición de igualdad simple:

$$A = a : \quad T' = \left\lceil \frac{TR}{IR.A} \right\rceil$$

donde  $TR$  es el total de registros en la tabla y  $IR.A$  es la cantidad de valores distintos que toma  $A$ . En promedio, cada valor  $a$  aparece  $\frac{TR}{IR.A}$  veces.

- Para una condición de igualdad compuesta:

$$A = a \wedge B = b : \quad T' = \left\lceil \frac{TR}{IR.A \times IR.B} \right\rceil$$

Se considera independencia entre  $A$  y  $B$ , por lo que la probabilidad de que un registro cumpla ambas condiciones es la multiplicación de sus selectividades.

- Para un rango superior:

$$A > a : \quad T' = \left\lceil \frac{TR \times dist(A_{\max}, a)}{dist(A_{\max}, A_{\min})} \right\rceil$$

Se estima qué fracción del dominio de  $A$  está por encima de  $a$  según su rango  $[A_{\min}, A_{\max}]$ , y se multiplica esa fracción por  $TR$ .

- Para un rango acotado:

$$a < A < b : \quad T' = \left\lceil \frac{TR \times dist(b, a)}{dist(A_{\max}, A_{\min})} \right\rceil$$

Similar al caso anterior, pero se estima la fracción de registros que caen entre  $a$  y  $b$ .

## 9. Planes de Ejecución y Optimización de Consultas

Una misma consulta (query) puede ejecutarse de diferentes maneras para obtener el conjunto de resultados deseado. Cada forma posible de ejecutarla se denomina **plan de ejecución** (*query plan*).

Un **plan de ejecución** (*query plan*) describe cómo el DBMS procesará una consulta paso a paso. Indica el orden de acceso a tablas, uso de índices, métodos de búsqueda y combinaciones, así como filtros y proyecciones.

Ahora, como encontrar el plan óptimo es un problema NP-completo, el modulo debe emplear otras técnicas para hallarlo en un tiempo razonable. Entre ellas tenemos:

- **Uso de heurísticas:** Se aplican transformaciones del álgebra relacional manteniendo los resultados obtenidos. Estas suelen mejorar la performance pero no siempre.
- **Estimación de selectividad:** Se hace uso de la información en la base de datos para estimar el grado de selectividad de la consulta (cantidad de tuplas devueltas). Esto permite darnos una idea de su costo.
- **Índices y tipo de archivo:** El plan elegido depende fuertemente de los índices dispuestos en las tablas y como se ordenan físicamente los datos en disco.

### Etapas del Procesamiento de una Consulta

El procesamiento de una consulta SQL en SQL Server consta de varias fases:

1. **Parse:** Verifica la sintaxis de la sentencia y la convierte en un árbol parseado.
2. **Bind:** Vincula referencias de objetos con metadatos y genera un árbol algebraico.
3. **Optimize:** Se generan y evalúan planes de ejecución alternativos; se elige el más eficiente.
4. **Execute:** Se ejecuta el plan seleccionado para obtener el resultado.

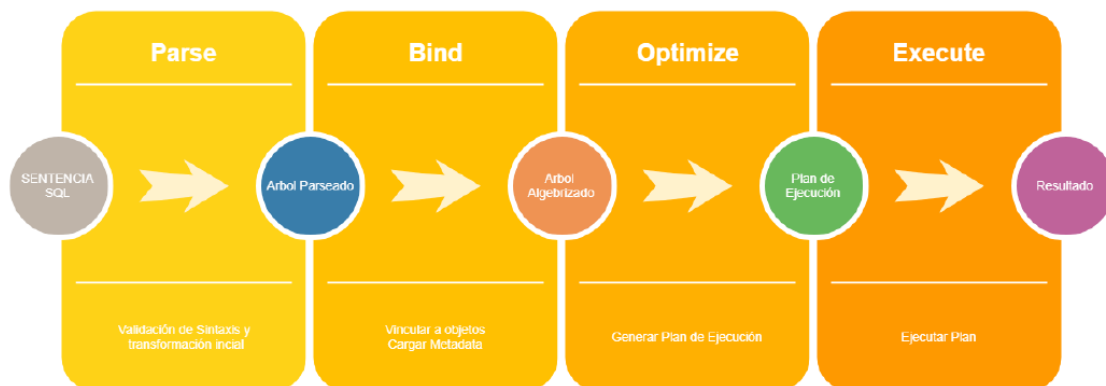


Figura 35: Fases del procesamiento de consultas

## Acceso a los Datos: Scan vs Seek

- **Index Seek:** Es una búsqueda dirigida en un índice. SQL Server utiliza las claves del índice para ir directamente a las filas que cumplen la condición. Es rápido y eficiente, especialmente cuando se utiliza una condición sobre una columna indexada (por ejemplo, con un `WHERE` o un `JOIN` bien optimizado).
- **Index Scan:** Es un recorrido completo del índice, fila por fila. Se utiliza cuando no hay filtros útiles o cuando la mayoría de las filas deben ser leídas. Es menos eficiente, pero a veces inevitable, especialmente si no hay índices adecuados o si se necesita leer toda la tabla.
- **Table Scan:** Similar al Index Scan, pero se aplica cuando no hay índice disponible. La tabla completa se recorre físicamente.






| Estructura         | Scan   | Seek  |
|--------------------|--|---|
| Heap               |  Table Scan           |   |
| Clustered Index    |  Clustered Index Scan |  Clustered Index Seek |
| Nonclustered Index |  Index Scan          |  Index Seek          |

Figura 36: Ejemplo de acceso a datos usando **Index Scan** (NonClustered)

### Lookup adicionales: RID y Key Lookup

Cuando se utiliza un índice que no contiene todas las columnas necesarias para satisfacer la consulta, SQL Server debe hacer una búsqueda adicional para completar los datos faltantes:

- **RID Lookup:** ocurre cuando se utiliza un índice *non-clustered* sobre una tabla **heap** (sin índice clustered). El índice contiene un puntero al **RID** (Row ID) en la tabla, y se realiza una operación adicional para buscar la fila completa.
- **Key Lookup:** ocurre cuando se utiliza un índice *non-clustered* pero la tabla tiene un **índice clustered**. En ese caso, el índice contiene la clave primaria (clustered key), y se utiliza esta clave para buscar la fila completa en el índice clustered.

Estas operaciones pueden ser costosas si se realizan muchas veces, ya que implican múltiples accesos a disco. Para evitarlas, se puede:

- Incluir las columnas necesarias directamente en el índice ('`INCLUDE`').
- Evaluar si conviene convertir el índice en un índice cubriente.

## Agregación en los Planes de Ejecución

Las operaciones de agregación aparecen explícitamente en el plan cuando se utilizan funciones como **SUM**, **COUNT**, **AVG**, **MIN**, **MAX**, o cláusulas como **GROUP BY** y **DISTINCT**. SQL Server puede usar distintas estrategias:

### Stream Aggregate

- Opera sobre datos **ordenados** por las columnas del **GROUP BY**.
- Procesa fila por fila hasta detectar un cambio en la clave de agrupamiento.
- Consume poca memoria y CPU, pero requiere ordenamiento previo.
- Se usa cuando los datos ya están ordenados o hay un índice útil.

### Hash Aggregate

- No necesita ordenamiento previo.
- Usa una tabla hash en memoria para agrupar.
- Es útil con grandes volúmenes de datos no ordenados.
- Puede usar TempDB si no hay suficiente memoria (spill).

### Sort + Distinct

- Se utiliza para consultas con **SELECT DISTINCT**.
- Primero se aplica un **Sort** sobre todas las columnas seleccionadas.
- Luego se eliminan duplicados.
- Es costoso si no hay un índice que permita ordenar eficientemente.

**Nota:** si existe un índice que cubre todas las columnas y mantiene el orden, puede evitarse el **Sort**.

## Operadores de Join en los Planes de Ejecución

### Nested Loops Join

- Para cada fila de la tabla externa (outer), se busca en la tabla interna (inner).
- Muy eficiente cuando la tabla externa tiene pocas filas y hay un índice útil sobre la inner.
- Costoso si ambas tablas son grandes y no hay índice.



## Merge Join

- Requiere que ambas tablas estén ordenadas por la clave de join.
- Avanza de forma secuencial comparando valores.
- Muy eficiente si ya están ordenadas o hay índices clustered.

**Nota:** si los datos no están ordenados, el plan incluirá un **Sort** previo, lo que puede encarecer la ejecución.

## Hash Join

- Crea una tabla hash en memoria con la entrada más pequeña .
- Recorre la entrada más grande buscando coincidencias.
- No necesita orden ni índices.
- Puede usar TempDB si no hay suficiente memoria (hash spill).

## Comparación de estrategias de Join

- **Nested Loops:** mejor cuando la outer es pequeña y hay índice sobre inner.
- **Merge Join:** ideal con datos ordenados por clave de join.
- **Hash Join:** bueno con tablas grandes y sin orden.

| Join Type      | Index on Joining Columns  | Usual Size of Joining Tables | Presorted | Join Clause |
|----------------|---|------------------------------|-----------|-------------|
| Hash           | Inner table: Not indexed<br>Outer table: Optional<br>Optimal condition: Small<br>outer table, large inner table | Any                          | No        | Equi-join   |
| Merge          | Both tables: Must<br>Optimal condition: Clustered<br>or covering index on both                                  | Large                        | Yes       | Equi-join   |
| Nested<br>loop | Inner table: Must<br>Outer table: Preferable  | Small                        | Optional  | All         |

Figura 37: Tabla comparativa

## Otros Operadores en los Planes de Ejecución

### Compute Scalar

- Evalúa expresiones escalares (aritméticas, funciones, conversiones, etc.) para cada fila procesada.
- Por ejemplo: calcular un total como `precio * cantidad`, o extraer año de una fecha con `YEAR(fecha)`.
- No filtra ni ordena; solo agrega columnas calculadas.

**Nota:** Puede parecer "inocuo", pero si la expresión es compleja o se evalúa sobre muchas filas, puede afectar el rendimiento.

### Filter

- Aplica una condición booleana a cada fila para decidir si debe seguir siendo procesada o descartada.
- Es usado cuando la condición no puede ser aplicada directamente en el acceso (por índice).
- Similar a un 'WHERE', pero separado del acceso físico.

**Nota:** Si el filtro depende de una función aplicada a una columna, no se puede usar un índice y el optimizador usará 'Filter'.

### UNION y UNION ALL

- El operador 'UNION' combina filas de dos o más conjuntos eliminando duplicados (incluye internamente un 'Sort' y 'Distinct').
- 'UNION ALL' hace lo mismo pero mantiene duplicados y es más eficiente.

## Consideraciones Avanzadas de Optimización

### Integridad Referencial y Planes de Ejecución

- Las claves foráneas permiten validar que los datos estén correctamente relacionados, pero también influyen en la optimización.
- Si el optimizador detecta que una clave foránea garantiza existencia o unicidad, puede evitar hacer ciertas validaciones o aplicar un 'Join Semi' en lugar de un 'Join Completo'.
- Esto mejora el rendimiento al reducir operaciones innecesarias.

## Hints

- Son instrucciones opcionales que se agregan en la consulta para influir en el plan de ejecución elegido.
- Pueden forzar el uso de un índice, de un tipo de 'join', desactivar la paralelización, etc.
- Útiles para debugging o cuando el optimizador no elige el mejor plan, pero deben usarse con precaución.

## Parameter Sniffing y OPTIMIZE FOR

- Cuando se ejecuta un procedimiento almacenado, el optimizador utiliza el primer valor recibido como parámetro para calcular el plan de ejecución.
- Esto se llama **Parameter Sniffing**. Puede generar planes eficientes para ese valor, pero subóptimos para otros.
- Ejemplo: una consulta con 'WHERE cliente\_id = @id' puede usar 'Seek' si '@id=1' pero 'Scan' si '@id=90000'.
- Solución: usar OPTIMIZE FOR UNKNOWN o OPTIMIZE FOR (@id = 123) para generar un plan más genérico o controlado.

## SARGable (Search ARGument-able)

- Una condición es **SARGable** si puede aprovechar un índice.
- Ejemplo SARGable: WHERE nombre = 'Juan' (puede usar índice).
- Ejemplo no SARGable: WHERE UPPER(nombre) = 'JUAN' (no puede usar índice).
- Reglas:
  - No aplicar funciones sobre columnas.
  - No usar operaciones aritméticas del lado izquierdo.
  - Usar operadores comparativos simples (=, >, <, BETWEEN).
- Hacer que las consultas sean SARGables mejora el rendimiento dramáticamente.

| SARGable predicates                         | Non-SARGable predicates                      |
|---|--|
| LastName = 'Clark' and FirstName = 'Steve'  | LastName <> 'Clark' and FirstName = 'Steve'  |
| LastName = 'Clark' and FirstName <> 'Steve' | LastName LIKE '%ar%' and FirstName = 'Steve' |
| LastName = 'Clark'                          | FirstName = 'Steve'                          |
| LastName LIKE 'C1%'                         |  |

Figura 38: Tabla SARGability

## Índices con Filtros y Columnstore

- Un **índice filtrado** es un índice parcial, que incluye solo las filas que cumplen una condición.
- Muy útil para tablas grandes con muchos valores nulos o poco frecuentes.
- Ejemplo:

```
CREATE INDEX idx_activos  
ON Clientes (estado)  
WHERE estado = 'Activo'
```

- También pueden combinarse con columnas incluidas (**INCLUDE**).
- Los índices **columnstore** se usan para consultas analíticas: son muy eficientes en lectura masiva, pero no tanto en escritura.

## 10. Bases de Datos Distribuidas (DDB)

Se denominan **bases de datos distribuidas** (DDB) a colecciones de bases de datos que, aunque están físicamente separadas y ubicadas en distintos nodos de una red de computadoras, se encuentran **lógicamente relacionadas** y funcionan como un solo sistema. Su gestión está a cargo de un **DDBMS** (Distributed DataBase Management System), cuyo objetivo principal es que la distribución sea **transparente para el usuario**, es decir, que las operaciones sobre los datos puedan realizarse como si todo estuviera centralizado.

La forma en que una consulta se ejecuta en un sistema distribuido depende de:

- La **topología y tipo de red** que conecta los nodos (LAN, WAN, conexión directa, nivel de conectividad).
- La **manera en que los datos están distribuidos** (fragmentación o replicación). Por ejemplo, en una operación de *join* es clave minimizar la cantidad de datos transferidos entre nodos.

### Características esperables

- **Disponibilidad y confiabilidad:** Se refieren a la probabilidad de que un sistema se encuentre continuamente disponible u operando en un determinado intervalo de tiempo (respectivamente, aunque los términos se usen de forma indistinta). Ante las fallas del sistema, un sistema confiable puede:
  - Enfatizar su tolerancia a las fallas, reconociendo que pueden ocurrir y diseñar mecanismos que las detecten y las remuevan antes de que ocurran.
  - Asegurar la ausencia de fallas en el sistema final mediante procesos de desarrollo que incluyen control de calidad y testing.

En el caso de los DDBMSs, estos deben tolerar fallos de sus componentes subyacentes sin alterar los pedidos de sus usuarios ni la consistencia de la base. El **RM** se encarga de tratar fallas en las transacciones, hardware y comunicaciones en las redes.

- **Escalabilidad y tolerancia a la partición:**
  - **Escalabilidad:** Medida en que un sistema puede expandirse mientras continúe operando ininterrumpidamente. Puede ser:
    - *Horizontal*: cantidad de nodos del sistema distribuido.
    - *Vertical*: capacidad de un nodo individual.
  - **Tolerancia a la partición:** Implica que el sistema pueda seguir operando aun cuando la red se particione.
- **Autonomía:** Medida en que los nodos individuales de una DDB puedan operar independientemente. Se busca que tenga alto grado. Existen varios tipos:
  - *Diseño*: Independencia del uso del modelo de datos/técnicas de gestión de transacciones entre los nodos.

- *Comunicación*: Medida en la que los nodos deciden compartir o no información con otros.
- *Ejecución*: Acciones “a gusto” del usuario.

## Ventajas de las DDBs

- Permiten mejorar sencilla y flexiblemente el desarrollo de aplicaciones, debido a la transparencia de control y distribución de los datos.
- Aumentan la disponibilidad al aislar sus fallas a su sitio de origen.
- Mejoran la performance al fragmentar y replicar los datos, dando nuevas oportunidades de conseguir escalabilidad horizontal (distribuyendo la carga entre nodos) y mantener cerca los datos necesitados (en nodos geográficamente cercanos a donde se realice la consulta).

## Teorema CAP

Publicado en 1998 y presentado por Eric Brewer en el año 2000, establece que cualquier sistema que comparta datos a través de la red puede cumplir con a lo sumo dos de las siguientes propiedades:

- **Consistency**: Toda lectura devuelve la escritura más reciente del ítem.
- **Availability**: Todo pedido recibe una respuesta, pero sin el ítem anterior, no se garantiza que esta respuesta sea la de la última escritura. Pero sí garantiza que no es errónea o un código de error.
- **Partition Tolerance**: El sistema sigue funcionando a pesar de la pérdida de una cantidad arbitraria de paquetes (mensajes) o nodos en la red (incluso aunque los pierda todos). La idea es que si decide seguir con la operación, puede perder consistencia por no disponer de la información que el nodo caído o incomunicable posee; o puede arriesgar disponibilidad si decide cancelar la operación o retrasarla (porque puede terminar retrasándola infinitamente).

Según Brewer, el objetivo del teorema debe enfocarse en encontrar el balance entre consistencia y disponibilidad mientras se encuentre la forma de manejar las particiones de la red, considerando cómo recuperarse ante eventuales fallas. Para lo segundo, el sistema debe detectar la partición, entrar en modo “particionado” y luego recuperar el modo anterior.

## Tipos de consistencia

- **Consistencia estricta o fuerte**: Un sistema tiene consistencia estricta cuando las lecturas retornan siempre el valor más reciente del ítem leído.
- **Consistencia débil**: Existe cuando el sistema no garantiza que una lectura obtenga un valor actualizado.

- **Read Your Own Writes (RYOW):** El cliente puede leer sus propias actualizaciones inmediatamente después de que hayan sido completadas, incluso si escribe en un servidor y lee desde otro. Las actualizaciones de otros clientes no tienen por qué ser visibles de inmediato.
- **Consistencia de sesión:** Más débil que RYOW. Garantiza RYOW solo si el cliente permanece dentro de la misma sesión, usualmente en el mismo servidor.
- **Consistencia causal:** Si un evento  $b$  es causado o influenciado por un evento previo  $a$ , la causalidad requiere que todos los procesos vean primero  $a$  y luego  $b$ . Escrituras relacionadas causalmente deben ser vistas por todos los procesos en el mismo orden. Las escrituras concurrentes pueden verse en un orden diferente en distintos nodos.
- **Lectura monótona (Monotonic Read):** Si un proceso lee el valor de un dato  $x$ , cualquier lectura posterior sobre  $x$  realizada por el mismo proceso devolverá siempre el mismo valor o uno más reciente.
- **Consistencia eventual:** Con el tiempo, las operaciones de lectura reflejan las escrituras. En equilibrio, el sistema devuelve el último valor escrito: a medida que  $t \rightarrow \infty$ , los clientes verán las actualizaciones más recientes.

## Propiedades BASE

Son las propiedades que cumplen las bases de datos NoSQL (de las cuales muchas incorporan funcionalidades de DDBs) y son mucho más laxas, o débiles, que las propiedades ACID (las cuales también son deseables en una DDB). Estas son:

- **Basic Availability:** El sistema debe garantizar disponibilidad en términos del teorema CAP incluso cuando ocurren fallas (si se produce una falla, en el peor de los casos la Base de Datos se reserva la posibilidad de particionarse). No obstante, esta disponibilidad no garantiza consistencia. Las lecturas pueden no leer la última escritura, y la escritura puede no persistir si se encuentran conflictos no reconciliables. Esto puede lograrse a través de una base de datos altamente distribuida, en donde no se guarde una sola copia con tolerancia a fallas sino múltiples réplicas en discos. De esa forma, una falla que inhabilita el acceso a un grupo de datos no necesariamente lo hace a toda la DB.
- **Soft-state:** El estado de la DB puede cambiar a lo largo del tiempo incluso sin intervención externa. Esto significa que en un momento dado no necesariamente habrá una única “versión” de cada dato y la consistencia debe ser manejada por los desarrolladores, no por el DBMS.
- **Eventual consistency:** El sistema garantiza que, posterior a la ejecución de escrituras, el sistema converge después de un tiempo no determinado a un estado donde cualquier lectura de ese ítem de datos siempre retorne el mismo valor. Nuevamente se recalca que los conflictos de concurrencia quedan en manos de los desarrolladores y no del DBMS.

## Fragmentación

Los fragmentos son las partes resultantes de dividir la base de datos en unidades lógicas. La fragmentación puede ser:

- **Horizontal (sharding):** Subdivisión en subconjuntos de tuplas de la relación original. Es análoga al operador **SELECT** y, si comprende a todas las tuplas, se denomina *completa* (puede ser disjunta). Su reconstrucción viene dada por **UNION**. Aumenta el rendimiento de las escrituras y lecturas al distribuir la carga en distintos nodos, pero reduce el rendimiento de operaciones analíticas al requerir **UNIONs** cuando se quieren todos los datos de la tabla.
- **Vertical:** Subdivisión en subrelaciones dadas por un subconjunto de columnas de la original. Es análoga al proyector y, si cada atributo está en al menos una proyección y todas comparten sólo la clave primaria, se denomina *completa*. La reconstrucción viene dada por **OUTER JOIN**. Aumenta el rendimiento de las lecturas y escrituras ya que distribuye la carga en distintos nodos cuando se trata de accesos a pocas columnas, pero a costa de que las operaciones que requieran el uso de todas las columnas se vean ralentizadas por la necesidad de realizar **JOINs**.
- **Mixta:** Combinación de las dos anteriores que puede reconstruir la relación con las operaciones en el orden apropiado.

Un **esquema de fragmentación** de una base de datos es un conjunto de fragmentos que incluye a todos sus atributos y tuplas, y permite reconstruirla con la secuencia de operaciones apropiada.

Por otra parte, un **esquema de asignación** describe la ubicación de los fragmentos en los nodos de la DDB y, si un fragmento está en más de un lugar, se denomina **replicado**.

## Replicación

Las réplicas son copias de los datos que permiten aumentar su disponibilidad y confiabilidad. Según su nivel, la replicación puede ser:

- **Total:** Todos los nodos tienen copias de la base de datos.
  - **Ventajas:** El sistema puede continuar operando si al menos uno está disponible y mejora el rendimiento de lecturas.
  - **Desventajas:** El rendimiento de escrituras empeora y las técnicas de control de concurrencia y recuperación son más costosas.
- **Nula:** Caso opuesto al anterior, en que ningún elemento está replicado.
  - **Ventajas:** Las escrituras son menos costosas y no requieren tanto control de concurrencia ni recuperación.
  - **Desventajas:** Tiene múltiples puntos de falla y cuellos de botella para las lecturas.



- **Parcial:** Algunos elementos se replican y otros no, distribuyéndose en sitios particulares. Esta depende de las metas de disponibilidad, rendimiento y el tipo de transacciones de cada sitio del sistema. La descripción de la replicación viene dada por el *esquema de replicación*.

## Control de concurrencia en DDBs

El control de concurrencia en bases de datos distribuidas se encarga de mantener la consistencia entre las copias de un mismo *data item*, intentando preservar las propiedades de las bases centralizadas. Para esto, se pueden usar distintos enfoques de **locking** que difieren según cuán centralizada o distribuida sea la toma de decisiones.

### ■ Esquemas centralizados

En estos enfoques, siempre existe un nodo designado que actúa como responsable principal de procesar las transacciones y administrar los **locks**.

- **Sitio primario:** Un único sitio es distinguido y procesa todas las solicitudes de transacción. Si otorga el **lock** a un nodo, este puede acceder a cualquier copia del dato. El DDBMS se encarga de propagar los cambios a todas las copias.
  - Ventajas: Extiende el modelo centralizado, y con *Two-Phase Commit (2PC)* la seriabilidad queda garantizada.
  - Desventajas: Existe un cuello de botella en el nodo primario y un único punto de falla que puede paralizar el sistema.
- **Sitio primario con backup:** Igual que el caso anterior, pero con un sitio de respaldo que toma el control si el primario falla.
  - Ventajas: Simplifica la recuperación ante fallos del primario.
  - Desventajas: El cuello de botella persiste y el rendimiento del locking se ve afectado.

### Elección de nuevo coordinador:

Ante una falla, en los casos sin backup las transacciones que acceden a los datos del sitio afectado deben ser abortadas y reiniciadas, mientras que de haber backup estas son suspendidas hasta ser designados estos sitios como primarios. Si ambos fallan puede iniciarse un proceso de elección de un nuevo coordinador desde un sitio *Y*:

1. Se considera fallido al coordinador si el nodo *Y* no logra comunicarse con él tras varios intentos.
2. *Y* envía a todos los nodos activos un aviso de que desea ser el nuevo coordinador.
3. Si recibe la mayoría de los votos, se lo designa coordinador (resuelve conflictos si más de un sitio desea el rol).

### ■ Esquemas distribuidos

En estos enfoques se evita un nodo central fijo, buscando balancear carga y mejorar disponibilidad.

- **Copia primaria:** Cada *data item* replicado tiene uno de sus nodos como *copia distinguida*, que administra los locks para ese ítem. El control se reparte por ítem, no por sistema completo.  
**Ventajas:** Elimina el cuello de botella y el punto único de falla del primario global.  
**Desventajas:** Mayor complejidad en el control de concurrencia y más tráfico de red para coordinar transacciones.
- **Votación (quorum):**  
 Cuando un nodo quiere un lock(X), consulta a **todas las copias** de X. Cada copia acepta o rechaza el pedido. Si se obtiene la mayoría de aprobaciones en un tiempo límite:
  - El nodo obtiene el lock y notifica a todas las copias que X fue tomado.
  - Si no logra mayoría, el lock se cancela y se informa el rechazo.**Ventajas:** Control de concurrencia totalmente distribuido (sin un nodo privilegiado).  
**Desventajas:** Mucho tráfico de mensajes y más complejidad si hay fallos durante la votación.

## Transparencia

La transparencia se basa en ocultar los detalles de implementación a los usuarios finales. Si es total, la visión del usuario es la de una base de datos centralizada. Existen distintos tipos de transparencia en bases de datos distribuidas:

- **Transparencia de organización de los datos:**
  - **Ubicación:** Permite ejecutar una tarea independientemente de dónde estén ubicados los datos.
  - **Nombres:** Un nombre asociado a un objeto es suficiente sin requerir datos adicionales para ubicarlo.
- **Transparencia de fragmentación:** El usuario no necesita conocer detalles sobre cómo están fragmentados los datos. Puede aplicarse a fragmentación horizontal, vertical, o ambas.
- **Transparencia de replicación:** El usuario no necesita conocer los detalles de replicación, ubicación ni motivo por el cual se replicaron los datos, aunque puede llegar a ser configurable.
- **Otras transparencias:**
  - **Diseño:** Oculta al usuario cómo está diseñada la base de datos entre los nodos.
  - **Ejecución:** Oculta al usuario dónde y cómo se procesan las consultas.

La transparencia tiene la ventaja de ofrecer una visión de la base de datos como si fuera centralizada, pero presenta el inconveniente de que no permite reflejar características deseables del DDBMS. Por lo tanto, la transparencia debe poder ser configurable, o debe alcanzarse un compromiso entre facilidad de uso y el sobre costo de proveerla.

## Catálogo

Es una base de datos que contiene la *metadata* del DBMS. Además de incluir información sobre las tablas, usuarios, vistas y demás objetos, en el caso del DDBMS también incluye información sobre:

- **Fragmentación:** Describe cómo se dividen las tablas en fragmentos (horizontales o verticales) para su distribución.
- **Asignación de fragmentos:** Indica en qué nodos o sitios de la red se almacena cada fragmento.
- **Replicación de datos:** Especifica qué fragmentos tienen copias en múltiples nodos y cómo se mantienen sincronizados.

Su administración debe contemplar autonomía de sitio, vistas, distribución y replicación. Existen tres esquemas posibles:

### Catálogo centralizado

Se almacena por completo en un sitio.

- **Ventajas:** Implementación sencilla.
- **Desventajas:**
  - Baja confiabilidad, disponibilidad y autonomía.
  - Centraliza la carga de procesamiento.
  - Los locks pueden producir cuellos de botella ante muchas escrituras.

### Catálogo totalmente replicado

Cada sitio posee una copia completa del catálogo.

- **Ventajas:** Puede responder consultas localmente.
- **Desventajas:** Requiere transmitir todas las actualizaciones a todos los sitios. Para mantener la consistencia se necesita un esquema centralizado tipo 2PC, lo que incrementa el tráfico de la red ante muchas escrituras.

### Catálogo parcialmente replicado

Cada sitio tiene información completa del catálogo de los datos almacenados localmente. Además, pueden tener en caché copias de entradas de otros sitios (no necesariamente actualizadas).

El sistema debe llevar un registro, para cada entrada del catálogo, de:

- Dónde se creó el objeto.
- Dónde están sus copias.

Esto permite propagar los cambios de las copias al original.

## Recuperación en DDBs

Este subsistema debe atender a dos principales problemas:

- **Fallos en sitios y comunicaciones:** A partir de un sitio  $X$  es difícil determinar si un sitio  $Y$  se encuentra caído sin intercambiar una gran cantidad de mensajes. En efecto, la falta de respuesta de  $Y$  podría deberse además a una falla en la comunicación que impida que haya recibido el mensaje o no pueda responder.
- **Commit distribuido:** Cuando una transacción actualiza un valor en varios sitios, no puede realizar el *commit* hasta asegurarse de que sus efectos no se perderán en ninguno de ellos (log). Para asegurar la correctitud frente a este problema, se suele usar el protocolo 2PC.

## Gestión de transacciones distribuidas

En las DDBs existen varios módulos encargados de garantizar las propiedades ACID:

- Gestor global de transacciones
- Gestor local de transacciones
- Gestor de control de concurrencia (CC)
- Gestor de recuperación

El gestor global actúa en el sitio de origen de la transacción y le pasa la operación al de CC. Este se encarga de manejar los *locks*, y hasta adquirirlo bloquea a la transacción. Tras obtenerlo, el *runtime processor* ejecuta la operación, luego libera el *lock* y le avisa al gestor de transacciones del resultado.

## Protocolo 2PC (Two-Phase Commit)

En este protocolo actúan los gestores de recuperación (global y local con su información correspondiente) y el *Transaction Manager* (TM) del nodo coordinador. Se divide en dos fases:

- **Primera fase:** El coordinador le pregunta a todos los nodos si están listos para realizar el commit (o abort) y espera recibir la mayoría de aprobaciones dentro de un tiempo de *timeout*. Si no las recibe, aborta la operación.
- **Segunda fase:** El coordinador les comunica a todos los nodos que se va a ejecutar el commit (o abort) y espera su respuesta positiva dentro de un nuevo *timeout*. Si no, cancela la operación. Sin embargo, si el coordinador falla tras solicitar el lock, los nodos restantes pueden quedar bloqueados permanentemente.

**Solución:** El protocolo 3PC agrega una fase intermedia de reconocimiento con timeout, que permite abortar si no hay respuesta tras la primera, o confirmar el commit antes de ejecutarlo. Esto evita que los nodos queden bloqueados ante la caída del coordinador.

## Gestión de consultas distribuidas

El procesamiento de consultas se divide en las siguientes etapas:

1. **Mapeo:** Se transforma la consulta SQL a álgebra relacional (AR) en base al esquema conceptual global. Luego se normaliza y reestructura como en el caso centralizado.
2. **Localización:** Se mapea la consulta resultante a múltiples fragmentos individuales, considerando la información de distribución y replicación de datos.
3. **Optimización de la consulta global:** Se selecciona una estrategia de ejecución entre varias candidatas cercanas a la óptima, según un criterio (tiempo, costo total, comunicación, etc.).
4. **Optimización de la consulta local:** Se ejecuta en todos los nodos de la DDB con técnicas de optimización centralizadas.

### Semijoin

Durante la tercera etapa, uno de los principales costos a considerar es la transferencia de datos. Una técnica útil para reducir la cantidad de tuplas a transferir es el **semijoin**.

- Consiste en enviar sólo las columnas necesarias para la junta a otro nodo, realizarla allí, y retornar solo las tuplas necesarias con sus atributos relevantes.

**Ventaja:** Si sólo una pequeña parte de la relación participa en la junta, se minimiza la transferencia de datos.

**Desventaja:** Es una heurística, y puede fallar si la estimación es incorrecta.

## Tipos de DDBs

Los sistemas de DDBs pueden diferir según el grado de:

### Homogeneidad

Se refiere a si todos los servidores y usuarios hacen uso del mismo software. La heterogeneidad puede surgir por diferencias en:

- **Modelo de datos:** Pueden diferir en su modelo (relacional, orientado a objetos, archivos, etc.). La misma información puede representarse como un atributo o como una relación. Es necesario un procesamiento de consultas inteligente basado en metadata.
- **Restricciones:** Varían de sistema a sistema y deben ser tratadas en el esquema global.
- **Lenguaje de consultas:** Incluso dentro del mismo modelo pueden existir diferentes versiones.

También puede haber heterogeneidad semántica si existen diferencias en el significado, interpretación y uso de los datos, producto de la autonomía de diseño. Se puede abordar mediante software que administre las consultas y transacciones desde la aplicación global hacia cada base de datos (y viceversa), como:

- *Middleware*
- *Application servers*
- *Enterprise Resource Planning (ERP)*
- Modelos y herramientas para integración e intercambio de datos
- Acceso basado en ontologías

### Autonomía local

Describe cuánto depende cada sitio del resto para funcionar como DBMS independiente. Existen tres dimensiones:

- **Comunicación:** Capacidad de decidir cuándo comunicarse con el resto.
- **Ejecución:** Capacidad de ejecutar operaciones sin interferencia externa y decidir su orden.
- **Asociación:** Capacidad de decidir cuánto compartir sus recursos y funcionalidad.

De esto surgen:

- **Federated DBMS (FDBS):** Cada servidor es autónomo, tiene usuarios locales, transacciones locales y un DBA. Existe un esquema global.
- **Peer-to-peer DBMS (p2pDBS):** Cada nodo se construye a medida que se necesita. Son heterogéneos y requieren un lenguaje canónico para traducir consultas.

## Arquitecturas paralelas y distribuidas

### Arquitecturas multiprocesador

- **Memoria compartida:** Múltiples procesadores comparten memoria principal y almacenamiento secundario.
- **Disco compartido:** Múltiples procesadores comparten disco, pero tienen memoria primaria propia.
- **Shared-nothing:** Cada procesador tiene su propia memoria y disco. Se comunican mediante redes de alta velocidad.

## Tipos particulares de arquitectura

- **Parallel DBMS (PDBMS):** Tipo de base de datos distribuida utilizada en la industria. Los nodos están físicamente cercanos y conectados por redes locales de alta velocidad. El costo de comunicación es bajo. Puede implementarse con cualquiera de las tres arquitecturas anteriores. Su principal diferencia con las distribuidas puras es su modo de operación.
- **Distribuidas puras:** Cada nodo está en un sitio distinto y se comunica por redes con costo no despreciable. Se basan en arquitectura *shared-nothing*, con un esquema conceptual global y un esquema interno por nodo. El catálogo global permite imponer restricciones y optimizar consultas.
- **Arquitectura 3-Tier:** Basada en cliente-servidor. Divide el sistema en tres capas:
  - **Presentation:** Interfaz con el usuario.
  - **Application:** Encargada de la lógica de negocio.
  - **Database Server:** Encargada de manejar, procesar y devolver las consultas y actualizaciones.

## Funcionalidades clave en la capa de aplicación (3-Tier)

- Generar un plan de ejecución distribuido y supervisar su ejecución.
- Asegurar la consistencia entre réplicas mediante control de concurrencia distribuido.
- Garantizar la atomicidad de las transacciones globales, ejecutando recovery global en caso de fallas.

**Nota:** Esta capa no siempre puede operar con *transparencia de distribución*, es decir, sin necesidad de especificar los sitios donde residen los datos requeridos por las consultas o transacciones.

## 11. Bases de Datos NoSQL

### Introducción General

Las bases de datos NoSQL (Not Only SQL) surgieron como una respuesta a las limitaciones de los modelos relacionales tradicionales frente al crecimiento exponencial de datos, la necesidad de escalabilidad horizontal, y la flexibilidad en la representación de la información. Se utilizan especialmente en aplicaciones web modernas, Big Data y sistemas distribuidos.


- **No requieren esquemas fijos:** admiten estructuras dinámicas.
  - **Alto rendimiento y escalabilidad horizontal:** diseñadas para distribuir la carga en múltiples nodos.
  - **No siguen necesariamente ACID:** se basan en modelos como BASE (Basically Available, Soft state, Eventually consistent).
  - **Modelos de datos variados:** documento, clave-valor, columnares y grafos.
- 

### Bases de Datos Orientadas a Documentos

Este modelo almacena datos en documentos generalmente estructurados como JSON o XML. La BD almacena y recupera documentos. Cada documento contiene pares clave-valor y puede representar una entidad completa.

#### Características principales

- **Modelo flexible:** cada documento puede tener su propia estructura, lo que favorece la evolución del esquema sin afectar al resto de la colección. Las colecciones agrupan documentos de naturaleza similar, pero sin requerir un esquema uniforme.
- **Alta afinidad con estructuras del mundo real:** ideal para aplicaciones web, APIs REST y representaciones jerárquicas de datos.
- **Consultas potentes:** permiten filtros, agregaciones, proyecciones, índices compuestos y texto completo. Los documentos pueden incluir listas, mapas anidados y estructuras complejas.
- **Escalabilidad horizontal:** permite particionar (sharding) los datos fácilmente.
- **Autonomía de documentos:** cada documento encapsula todos los datos necesarios para una entidad, evitando la necesidad de joins.



```
graph LR; A[Modelo Conceptual] --> B[DID]; B --> C[Documentos];
```

Modelo Conceptual -> DID -> Documentos

Figura 39: Diseño



- **DER** Modelo conceptual de alto nivel.
- **DID** (Modelo/Diagrama de Interrelación de Documentos).
- **JSON Schema** especificación de la estructura de los documentos.

## Desnormalización

La **desnormalización** consiste en **referenciar o incrustar datos dentro de un mismo documento** en lugar de separarlos en múltiples colecciones relacionadas, como ocurriría en un modelo relacional normalizado.

La **desnormalización parcial** consiste en un enfoque intermedio donde solo se **incrustan o duplican** los datos más consultados juntos, mientras que otros se mantienen como **referencias** para reducir redundancia y mantener consistencia. Se usa para equilibrar rendimiento en lecturas con un tamaño de documentos controlado.

- **Referenciar:** En un documento se hace referencia a un ID o una lista de ID de otro documento.
- **Incrustar:** En un documento se incluyen todos los datos (en principio) de otro documento

## Cuándo se genera un documento adicional

- Para representar **relaciones complejas** como ternaria.
- Para optimizar **consultas muy frecuentes**.
- Cuando existen **muchas mediciones que se actualizan constantemente**.

## Ventajas:

- No requiere joins ni operaciones complejas de relación.
- Ideal para contenidos semiestructurados y de rápida evolución.
- Alta eficiencia en lecturas rápidas y consultas de documentos completos.
- Esquema dinámico y fácil de modificar.
- Adecuado para aplicaciones con estructuras jerárquicas o anidadas.

### Desventajas:

- Posible redundancia y duplicación de datos.
- No se controla integridad referencial ni consistencia entre documentos.
- Las actualizaciones de datos duplicados deben realizarse manualmente.
- El rendimiento de algunas operaciones complejas puede verse afectado.

### Casos de uso comunes:

- Gestión de perfiles de usuario.
  - Carritos de compra y catálogos de productos.
  - Aplicaciones móviles y web en tiempo real.
  - Contenidos personalizados, redes sociales, blogs y foros.
- 

## Bases de Datos Column-Family

Las bases de datos de Column-Family están diseñadas para manejar grandes volúmenes de datos distribuidos con alta disponibilidad y escalabilidad horizontal, su diseño se basa en las consultas y flujo de la aplicación. Almacenan los datos en columnas (a diferencia de las otras BD que lo hacen por filas).

### Principales diferencias con el modelado relacional

- El diseño del esquema parte de las **consultas y flujos de la aplicación**, no solo del modelo de datos conceptual.
- Se prioriza que cada consulta acceda a una única partición (*partition per query*) para maximizar la eficiencia.
- El espacio de almacenamiento es secundario frente al rendimiento de lectura y escritura.

### Componentes y claves en bases Column-Family

- **Namespace:** Es un agrupamiento lógico de tablas. Permite organizar distintas colecciones de datos dentro de un mismo sistema.
- **Tablas:** Cada tabla consiste en múltiples filas, organizadas según claves de partición y clustering.
- **Row (Fila):** Una fila se estructura como un conjunto de **familias de columnas**.
- **Column Family:** Agrupa columnas relacionadas como un contenedor flexible de filas.

- **Row/Primary Key (Clave primaria):** Es la clave completa que identifica de manera única una fila en la tabla. Siempre está compuesta por:

$$\text{Primary Key} = \text{Partition Key} + \text{Clustering Keys}.$$

Incluye la clave de partición (obligatoria) y, opcionalmente, las claves de clustering que ordenan las filas dentro de cada partición. A veces se le dice “Row Key” a toda la Primary Key (Partition + Clustering), pero en otras fuentes se usa “Row Key” solo para la Partition Key. (Si entendí bien en las clases prácticas, Row Key = Primary Key, pero no lo sé :( )

- **Partition Key:** Es uno o más atributos (clave simple o compuesta) que determinan **en qué nodo del clúster se almacena la fila**. Todas las filas con la misma Partition Key se guardan en la misma partición física.
- **Clustering Keys:** Columnas opcionales que vienen después de la Partition Key en la Primary Key. Sirven para:
  - Ordenar las filas dentro de cada partición (ASC o DESC).
  - Permitir búsquedas por igualdad o por rango (>, <, BETWEEN).
  - Garantizar unicidad (por ejemplo, un UUID como última clave de clustering).

**Densidad:** Densidad baja. En una fila, cada familia puede tener cualquier número de columnas, incluyendo la posibilidad de no tener ninguna.

## Ventajas y desventajas

### Ventajas:

- Alto rendimiento en lecturas por rangos.
- Escalabilidad horizontal y tolerancia a fallos
- Flexibilidad para datos esparcidos (*sparse*) y consultas basadas en particiones.

### Desventajas:

- Mayor complejidad para modelar relaciones complejas.
- No existen joins ni integridad referencial automática.
- Restricciones en operadores de consulta: sólo sobre claves de partición y clustering.

## Metodología de diseño para Cassandra (Chebotko et al., 2015)

La propuesta de Chebotko introduce una metodología de tres pasos para modelar datos en Cassandra:

1. **Modelado conceptual y del flujo de aplicación:** Construcción de un diagrama entidad-relación (ER) y de un diagrama de flujo de aplicación que identifique patrones de acceso, atributos de búsqueda, ordenamientos y agregaciones.
2. **Modelado lógico (núcleo del proceso):** Conversión del modelo conceptual en un esquema lógico para Cassandra, aplicando los siguientes principios (DMP):
  - **DMP1:** Conocer los datos (entidades, claves, cardinalidades).
  - **DMP2:** Conocer las consultas y rutas de acceso.
  - **DMP3:** Uso de **anidamiento de datos** mediante particiones múltiples o colecciones.
  - **DMP4:** **Duplicación de datos** para soportar múltiples consultas.

Además, se aplican **en orden** las siguientes **Reglas de Mapeo (MR)**:

- **MR1 (Entidades y relaciones):** Cada tipo de entidad o relación se mapea a una tabla; las instancias se representan como filas y los atributos, como columnas. Las claves conceptuales que identifican a las entidades o relaciones deben preservarse en la tabla.
- **MR2 (Atributos de búsqueda por igualdad):** Los atributos usados en condiciones de igualdad se mapean primero como **partition key** (K).
- **MR3 (Atributos de búsqueda desigualdad):** Los atributos usados en condiciones de desigualdad o rangos se mapean como **clustering keys**, después de las columnas usadas en igualdad (MR2).
- **MR4 (Atributos de ordenamiento):** Los atributos que determinan el orden de los resultados se mapean como **clustering keys**, configurando el orden ascendente (ASC) o descendente (DESC) en la definición de la tabla, después de las columnas usadas en desigualdad (MR3).
- **MR5 (Atributos clave):** Los atributos que identifican de forma única cada fila deben formar parte de la **clave primaria**, después de las clustering keys de igualdad y rango. Esto asegura que cada fila pueda identificarse de manera unívoca.

Primary Key:  $K(\text{MR2}) \rightarrow C \uparrow (\text{MR3}) \rightarrow C (\text{MR4}) \rightarrow C \uparrow (\text{MR5})$ .

Respetar este orden es obligatorio: una violación puede hacer que la consulta no pueda ejecutarse en Cassandra.

3. **Modelado físico y optimización:** Ajuste del modelo lógico considerando tamaños de partición, agregaciones, índices invertidos, optimización de concurrencia y limitaciones del motor.

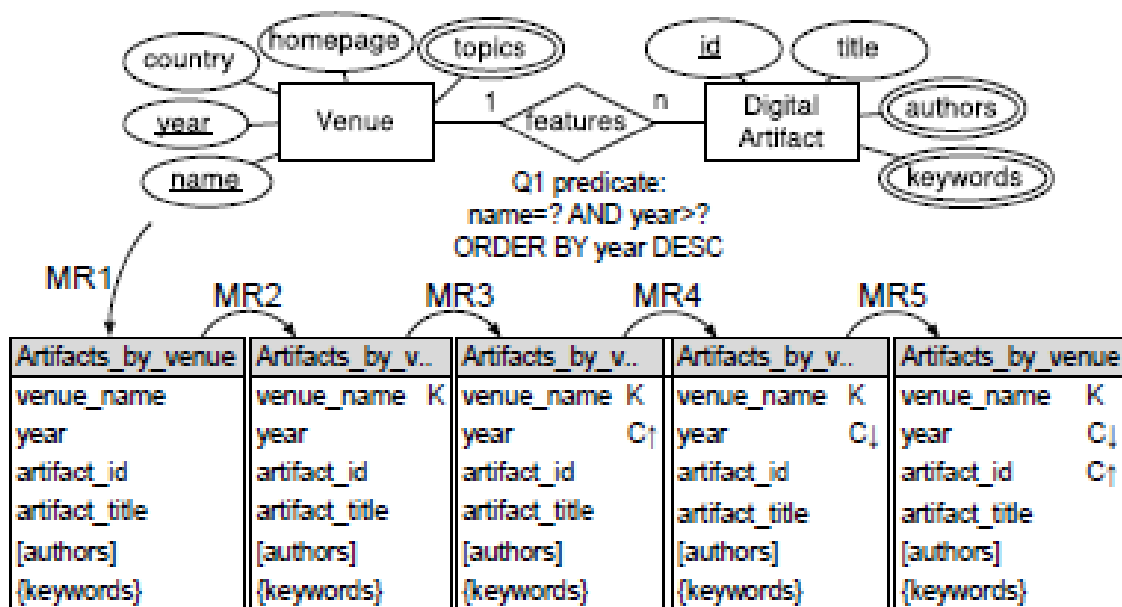


Figura 40: Ejemplo

### Ejemplo con partición dummy

Si una consulta no filtra por ningún atributo de partición (por ejemplo, listar todos los usuarios activos), se puede usar un **dummy** para evitar que cada fila caiga en una partición diferente:

```
CREATE TABLE usuarios_activos (
  dummy TEXT,
  user_id UUID,
  nombre TEXT,
  PRIMARY KEY (dummy, user_id)
);
```

#### Consulta:

```
SELECT nombre FROM usuarios_activos WHERE dummy = 'global';
```

De esta forma, todos los datos se agrupan en una sola partición, facilitando escaneos completos sin múltiples lecturas de particiones.

## Bases de Datos Clave-Valor

Son bases de datos que funcionan como un diccionario. Almacenan pares clave-valor, donde la clave es única y el valor es un bloque de datos. Son las más simples y rápidas. opera a través de la clave. y cada valor asociado a una clave puede tener diferente tipo, y puede cambiar libremente.

- Acceso ultra rápido por clave.
- No permiten búsquedas por contenido del valor.
- Escalables, simples y veloces.
- Ideales para caché, almacenamiento de sesión o datos temporales.

### Claves

- Las claves deben ser significativas y no ambiguas.
- Se recomienda el uso de un delimitador común como `:`.
- Es buena práctica mantener las claves cortas, pero sin perder expresividad.
- No hay columnas ni estructura interna del valor; la semántica se codifica en la clave.

### Namespace

Pueden surgir si varias aplicaciones usan la misma clave para diferentes propósitos. Se recomienda el uso de espacios de nombres(Namespace) para prevenirlo.

- Evita conflictos entre aplicaciones.
- Se puede incluir como prefijo en la clave.
- Por ejemplo: `ConcertApp:ticketLog:9888`

### Tiempo de vida (TTL)

Las claves pueden tener tiempo de expiración, útil por ejemplo para reservar asientos temporalmente durante una compra. Comandos como `SET EX 60` permiten definirlo directamente.

## 12. BigData

**Big Data** es un término que hace referencia a conjuntos de datos tan grandes y complejos que precisan de aplicaciones informáticas no tradicionales de procesamiento para tratarlos adecuadamente. En general, se enfoca en el conjunto de herramientas que permiten:

- Revelar relaciones y dependencias.
- Realizar predicciones de resultados y comportamientos.

**Big Data** se los define a los datos que cumplen con las 5 V's en mayor o menor medida.

- **Volumen:** presentan un gran volumen de datos.
- **Velocidad:** accesibles a velocidades que permiten el acceso en cuestión de milisegundos.
- **Variedad:** con datos de gran variedad, de muchos entornos distintos y diversos, que pueden estar estructurados o no, venir de paradigmas varios, e integrar muchos modelos.
- **Veracidad:** donde los datos pueden ser poco confiables, ruidosos, mezclar orígenes de datos que al combinarlos, su output puede generar incertidumbre, y que por tanto es necesario tratarlos, analizarlos, categorizarlos según fiabilidad, con el fin de poder corroborar con algún criterio su Veracidad.
- **Valor:** teniendo en cuenta que los datos proveen valor intrínseco al permitir tomar decisiones eficientes y precisas que le otorgan Valor.

### Repercusiones

- Se abandonó el paradigma “one size fits all”.
- Aparecen arquitecturas especializadas para distintos tipos de problemas.
- Se necesitan nuevos modelos de datos y lenguajes de consulta.
- Surgen tecnologías como “shared nothing”, alta disponibilidad, “hot standbys”, multi-threading, etc.

### Dos aspectos clave:

1. Soporte para entradas con incertidumbre.
2. Soporte para salidas con incertidumbre.

## 13. Data Mining

**Data Mining** es el proceso de extracción de patrones o información interesante (no trivial, implícita, previamente desconocida y potencialmente útil) a partir de grandes volúmenes de datos.

### Funcionalidades del Data Mining

1. **Descripción:** Caracterización y discriminación.
2. **Asociación:** Reglas del tipo IF-THEN, unidimensionales o multidimensionales.
3. **Clasificación y predicción:** Modelos que distinguen clases o predicen valores futuros.
4. **Clustering:** Agrupación de datos sin clase conocida.
5. **Análisis de outliers:** Detección de anomalías o fraudes.
6. **Análisis de tendencias y evolución:** Patrones secuenciales, regresión, análisis de similitud.

### Outliers y Precisión del Modelo

Los **outliers** son observaciones que siguen un comportamiento diferente al del resto en una o más variables. Si bien pueden utilizarse para detectar anomalías, en general pueden distorsionar los resultados, por lo que suele ser conveniente descartarlos.

### Origen de la Variación en los Datos

La variación que da origen a los *outliers* puede provenir de:

- **La fuente:** Proviene de las observaciones mismas y representa un comportamiento natural en relación con cierta variable de estudio.
- **El medio:** Deriva de un uso incorrecto de la técnica de medición o de la falta de precisión en la valoración. Incluye, por ejemplo, redondeos forzosos en variables continuas.
- **El experimentador:**
  - **Error de planificación:** Cuando no se delimita correctamente la población o se realizan observaciones que no corresponden.
  - **Error de realización:** Se produce al valorar incorrectamente los elementos (transcripciones erróneas, lecturas equivocadas de instrumentos, etc.).

Se considera que una **observación atípica** proviene de la variación del primer tipo, mientras que una **observación errónea** se asocia con las otras dos. Ambas pueden ser *outliers*, y conviene analizarlas cuidadosamente antes de decidir su eliminación.



## Precisión del Modelo

Para estimar la precisión de un modelo, es necesario construir dos conjuntos de datos:

- **Conjunto de entrenamiento:** Usado para ajustar el modelo.
- **Conjunto de testeo:** Distinto e independiente del anterior, utilizado para validar la generalización del modelo.

La **independencia** entre ambos conjuntos es clave para evitar **overfitting**.

La **precisión del modelo** se mide con los resultados sobre el conjunto de test, pudiendo construirse una **matriz de confusión**. En problemas de decisión binaria, esta matriz permite distinguir entre: **Verdaderos positivos y verdaderos negativos** (resultados correctos, ubicados en la diagonal), **Falsos positivos y Falsos negativos**.

## Tipos de Técnicas

### Algoritmos Supervisados

- **Definición:** Aprenden a partir de un conjunto de datos **etiquetados**, donde cada ejemplo incluye las características de entrada (features) y la salida esperada (etiqueta o valor numérico).
- **Objetivo:** Encontrar una función que relacione las entradas con las salidas correctas, para poder predecir resultados en datos nuevos.
- **Tareas típicas:**
  - **Clasificación:** predecir una categoría (ej.: correo *spam* o no *spam*).
  - **Regresión:** predecir un valor numérico (ej.: precio de una vivienda).
- **Ejemplos:**
  - Árboles de decisión
  - Redes neuronales
  - Regresión (lineal y logística)

### Algoritmos No Supervisados

- **Definición:** Trabajan con datos **sin etiquetar**; no cuentan con una salida o resultado predefinido.
- **Objetivo:** Descubrir **estructuras, patrones o relaciones ocultas** en los datos.
- **Ejemplos:**
  - Clustering jerárquico y no jerárquico
  - Reglas de asociación

# Algoritmos Supervisados

## Redes Neuronales

- Imitan el funcionamiento del cerebro humano.
- Son capaces de:
  - Aprender patrones.
  - Adaptarse a cambios y ruido.
  - Resolver problemas complejos.
- Limitaciones: no aptas para cálculos precisos o procesamiento en serie.

## Entrenamiento

El entrenamiento de una RNA (Red Neuronal Artificial) sigue una **regla delta generalizada**, consistente en un proceso iterativo con todos los datos de entrenamiento, que puede repetirse varias veces. El procedimiento es el siguiente:

1. Calcular la diferencia entre la salida resultante y la esperada.
2. Corregir los valores de las entradas para achicar las diferencias, en base a una constante delta muy pequeña.

Este método busca que la diferencia entre la salida obtenida y la esperada se minimice progresivamente. Si se hiciera una corrección demasiado grande, se podría modificar en exceso lo ya aprendido por la red, afectando su desempeño.

## Tipos de Redes Neuronales

Entre los tipos de redes neuronales artificiales más utilizadas se encuentran:

- **Redes Neuronales Profundas (Deep Neural Networks):** Es un término paraguas para muchos tipos de redes neuronales que poseen múltiples capas ocultas, otorgando gran profundidad al modelo. Su principal problema es el elevado costo computacional que implica entrenarlas, debido a la gran cantidad de nodos y conexiones que requieren ser optimizados.
- **Redes Neuronales Convolucionales (CNN - Convolutional Neural Networks):** Son más sencillas y están basadas en la operación matemática de convolución. Estas redes buscan percibir detalles de más alto nivel y suelen utilizarse en tareas de detección de patrones en imágenes, como por ejemplo bordes, texturas o formas.

## Árboles de Decisión

Son modelos en forma de árbol que se utilizan para clasificar una entrada desconocida según sus atributos.

- Representan decisiones mediante nodos internos y hojas.

- Fácil interpretación humana.
- Cada camino desde la raíz hasta una hoja en un árbol de decisión puede traducirse en una regla del tipo IF-THEN.
- Riesgo de **overfitting**, mitigado con:
  - **Prepruning**: detener crecimiento temprano.
  - **Postpruning**: poda de ramas innecesarias.

## Regresión

- **Lineal**: predice una variable continua basada en otras.
- **Logística**: predice probabilidad de clases dicotómicas.
- Supuestos: relación lineal entre las variables, independencia de errores, varianza de errores constante, linealidad, error esperado cero.

## Clasificación

1. **Entrenamiento**: se construye un modelo a partir de un conjunto de datos etiquetados.
2. **Evaluación**: se mide la precisión usando un conjunto de test.
3. **Matriz de confusión**: permite calcular métricas como:
  - TPR (sensibilidad), TNR (especificidad), Precisión, Accuracy.

## Algoritmos No Supervisados

### Clustering

- Agrupa datos sin etiquetas conocidas.
- Un buen clustering maximiza la similitud intra-cluster y la diferencia entre clusters.
- **Jerárquico**:
  - AGNES (aglomerativo) y DIANA (divisivo).
  - Enlaces: simple, completo, promedio, Ward, centroide.
- **No jerárquico**:
  - k-means, k-medoids (PAM).
  - Más rápido pero requiere definir el número de clusters.

### Distancias y Similitud

- **Euclídea, Manhattan, Minkowski**: aplicables a variables numéricas.

- **Binarias:** Jaccard, coeficiente simple.
- **Nominales:** macheo simple o variables dummy.
- **Ordinales:** pueden tratarse como numéricas con normalización.

## Reglas de Asociación

- Encuentran relaciones frecuentes entre ítems en bases transaccionales.
- Ejemplo: Si se compra B, entonces se compra C.
- **Medidas:**
  - **Soporte:** frecuencia de aparición.
  - **Confianza:** probabilidad condicional.
  - **Lift (mejora):** capacidad predictiva.
- **Tipos de reglas:**
  - Booleanas o cuantitativas.
  - Unidimensionales o multidimensionales.
  - Con o sin jerarquías.

## Aplicaciones Actuales

- **Text mining:** clasificación de textos, análisis de sentimientos.
- **Análisis de comunidades:** detección de grupos afines.
- **Análisis de trayectorias:** movimiento de aves.

## 14. Gobierno de Datos

**Gobierno de Datos** es el desarrollo y ejecución de **arquitecturas, prácticas y procedimientos** que manejan adecuadamente las necesidades del **ciclo de vida de datos** de una empresa.

Esto incluye aspectos de **calidad, arquitectura, seguridad y metadata** de los datos, y comprende a toda la organización, no sólo al sistema en sí, ya que los datos son considerados un activo de la empresa.

### Nivel de madurez del Gobierno de Datos:

- **Indisciplinado:** Las decisiones de negocio dependen de la tecnología. Los datos pueden ser inconsistentes o duplicados, y hay poca flexibilidad para adaptarse a cambios en el negocio.
- **Reactivo:** El negocio comienza a influir en las decisiones tecnológicas. Sin embargo, la información sigue siendo redundante y poco controlada, generando altos costos en el mantenimiento de múltiples aplicaciones.
- **Proactivo:** Los equipos de negocio y tecnología trabajan colaborativamente. Los datos son considerados un activo estratégico de la compañía.
- **Gobernado:** Los modelos de negocio guían las decisiones tecnológicas. Existen procesos estandarizados para la gestión de los activos de datos. Las decisiones corporativas se basan en datos certeros, generando beneficios tangibles gracias a la aplicación del programa de gobierno.

### Calidad de Datos

Tener los datos almacenados no garantiza su validez permanente. La **calidad de los datos** exige un monitoreo constante, que requiere inversión de tiempo y recursos. Existen diversos tipos de errores que pueden surgir:

- Valores fuera de rango.
- Falta de estandarización.
- Datos inválidos.
- Diferencias culturales.
- Discrepancias en el formato.
- Errores cosméticos.
- Inconsistencias provenientes de la metadata.
- Otros.

Para analizar la calidad de los datos se pueden aplicar diferentes técnicas:

- **Análisis univariado:** Evaluación de cada variable individualmente. Incluye obtener el valor mínimo, máximo, media, mediana, moda, histogramas y otras estadísticas descriptivas.
- **Análisis bivariado:** Estudia relaciones entre pares de variables, mediante coeficientes de correlación, tablas de contingencia, diagramas de dispersión, entre otros.
- **Perfilado de los datos:** Consiste en analizar la información en cada sitio para identificar posibles inconsistencias.

## Arquitectura de Datos

La **arquitectura de datos** es un conjunto de **especificaciones que ayudan en la estandarización** de cómo una organización recibe, almacena, transforma, distribuye y utiliza los datos. Esto contribuye a optimizar las inversiones en datos.

### Características de una Buena Arquitectura de Datos

- **Colaborativa:** Todas las áreas deben colaborar.
- **Administrada:** Buena gobernanza de los datos.
- **Simple:** Minimiza la variedad de herramientas utilizadas, reduce la duplicación de datos y facilita su uso.
- **Elástica:** Capaz de adaptarse a demandas crecientes o cambiantes.
- **Segura:** Con políticas de seguridad adecuadas.
- **Resiliente:** Alta disponibilidad y capacidad de recuperación ante fallas.

### Data as a Service (DaaS)

**Data as a Service (DaaS)** es una estrategia de administración de datos en la que una compañía renta su capacidad de almacenamiento, integración, procesamiento y servicios de analítica a otras compañías o terceros mediante servicios en la nube.

### Integraciones de Bases de Datos

Las bases de datos pueden integrarse entre sí por razones como fusiones de compañías o federaciones. Para facilitar este proceso se emplean herramientas que siguen distintas estrategias:

- **ETL (Extract - Transform - Load):** Los datos se transforman en un área intermedia (*staging area*) antes de cargarlos a la *Data Warehouse*, donde se almacenarán para su análisis.

- **ELT (Extract - Load - Transform):** Los datos se cargan directamente a la *Data Warehouse* y allí se transforman. Esto permite reutilizar los datos transformados en el futuro sin volver a cargarlos, aunque cualquier nuevo dato requiere una nueva transformación.

**Nota:** La fase que se realiza en segundo lugar tiene prioridad sobre la tercera.

## Seguridad en Base de Datos

La **seguridad** en bases de datos abarca diversas dimensiones para proteger los datos frente a pérdidas, accesos no autorizados o fallos del sistema.

Se puede analizar desde dos ejes principales: los **aspectos** que busca garantizar y los tipos de **mecanismos** aplicados.

### Aspectos clave de la seguridad:

- **Integridad:** garantiza que los datos almacenados sean correctos, completos y coherentes. Protege frente a modificaciones indebidas o accidentales, y se apoya en restricciones, validaciones y controles de acceso.
- **Confidencialidad:** asegura que solo los usuarios autorizados puedan acceder a determinada información. Esto implica la utilización de mecanismos como cifrado, roles de usuario, políticas de privacidad y autenticación.
- **Disponibilidad:** se refiere a que los datos estén accesibles y utilizables por los usuarios autorizados cuando los necesiten. Incluye medidas frente a caídas del sistema, ataques de denegación de servicio (DoS) y planificación de respaldos.

### Tipos de seguridad aplicables

- **Seguridad física:** Medidas destinadas a proteger el hardware y la infraestructura donde se almacenan los datos. Ejemplos: control de acceso a servidores, vigilancia, protección contra incendios y sistemas de respaldo eléctrico.
  - **Vulnerabilidades:** fallos o ataques sobre *hardware* como PCs, servidores o dispositivos IoT.
- **Seguridad lógica:** Controles implementados mediante software para proteger el acceso y uso de la información. Incluye contraseñas seguras, cifrado, control de accesos, firewalls, auditorías y autenticación multifactor.
  - **Vulnerabilidades:**
    - **Software:** fallos o ataques en back-end, base de datos, front-end o navegadores.
    - **Red:** interceptación de tráfico, *sniffing* o captura de paquetes.

- **Seguridad administrativa:** Políticas, normas y procedimientos definidos por la organización para garantizar un manejo seguro de la información. Ejemplos: capacitación del personal, definición de roles y permisos, auditorías internas y planes de contingencia.
  - **Vulnerabilidades:** errores o malas prácticas en el entorno humano o administrativo.

**Dilema:** Mayor seguridad implica menor comodidad.

## Reglas de Privilegios

Los privilegios se otorgan mediante reglas que pueden clasificarse en:

- **Privilegios de Sistema:** Permiten el acceso a comandos u operaciones específicas de la base de datos.
- **Privilegios de Objetos:** Permiten al usuario acceder de forma específica a determinados objetos (por ejemplo, tablas o vistas).

## Controles de Acceso

Existen diferentes formas de definir políticas de acceso, conocidas como **Controles de Acceso**:

- **Discretionary Access Control (DAC):** Los privilegios son otorgados según el usuario y el objeto accedido. El “dueño” del objeto es quien se responsabiliza por definir los permisos.
- **Role-based Access Control (RBAC):** Los privilegios se otorgan según roles. Un usuario puede tener uno o más roles, y un objeto puede exigir uno o más roles para ser accedido.
- **Rule-based Access Control:** Los privilegios se otorgan en función de un conjunto de reglas predefinidas.
- **Mandatory Access Control (MAC):** Los privilegios se otorgan según rangos jerárquicos de roles. Para acceder a un dato, el usuario debe poseer el rango adecuado.

## Monitoreo de Comportamiento Sospechoso

Es posible realizar **logging y monitoreo** del comportamiento de los usuarios.

Se puede optar por hacer **monitoreo completo** de todas las acciones de los usuarios, que implica muchos costo de almacenamiento.

O **monitoreo específico** de comportamientos considerados sospechosos. Ejemplos de comportamientos sospechosos incluyen:

- Logins exitosos o fallidos.



- Accesos exitosos o fallidos a la base de datos.
- Uso de la base de datos en horarios no habituales.
- Múltiples intentos de acceso desde distintos usuarios en una misma terminal.

## Seguridad del Backend vs de la Base de Datos

Principio del mínimo privilegio: cada usuario o proceso debe tener solo los permisos necesarios para hacer su tarea y nada más.

### Roles en el backend

- **Transparencia:** los administradores deben poder ver lo mismo que sus subordinados.
- El rol de superusuario no debe estar activo todo el tiempo.
- Las acciones deben estar **auditadas**.

### Las contraseñas

- Deben ser privadas (ni siquiera el admin debe poder verlas).
- Almacenarlas con hash fuerte.
- No incluirlas en el código ni en repositorios.
- Evitar contraseñas de prueba en producción.
- Usar archivos de configuración protegidos.
- Tener políticas de contraseñas claras.

### El control de usuarios

- Usar módulos estándar.
- Considerar Single Sign-On (SSO).
- Verificar estado y rol del usuario durante toda la sesión.
- Registrar accesos, intentos fallidos, estadísticas.

### Inyección de código

- Separar lo fijo (estructura) de lo variable (valores).
- Aplicar sanitización o quoting en un único punto.
- Centralizar contacto con otros sistemas (BD, HTML, shell).

## 15. Data Mesh y sus principios

**Data Warehouse** es una base de datos orientada al análisis que integra y consolida datos históricos y actuales provenientes de múltiples fuentes de información. Está optimizado para consultas y generación de reportes, utilizando un esquema estructurado (normalmente modelos en estrella o copo de nieve) para soportar procesos de *Business Intelligence* y toma de decisiones estratégicas.

**Data Lake** es un repositorio centralizado que almacena grandes volúmenes de datos en su formato original (estructurados, semiestructurados y no estructurados). Permite el procesamiento flexible de datos para análisis avanzados, aprendizaje automático y exploración, sin requerir un esquema fijo al momento de la ingesta (*schema-on-read*).

**Data Mesh** es un **enfoque descentralizado** para la gestión y análisis de datos a gran escala. Surge como alternativa a los Data Lakes o Data Warehouses centralizados, buscando **distribuir la responsabilidad sobre los datos a los equipos de negocio**, tratándolos como productos con calidad y accesibilidad garantizadas.

### Cuatro principios fundamentales de Data Mesh

1. **Propiedad de los datos por dominios (Domain Ownership):** Cada equipo o dominio de negocio (ventas, marketing, logística, etc.) es responsable de gestionar, documentar y asegurar la calidad de sus datos. Se elimina la dependencia de un equipo central único para todo el ciclo de vida de los datos.
2. **Datos como producto (Data as a Product):** Los conjuntos de datos se tratan como productos internos, con responsables definidos, documentación, estándares de calidad y métricas. Los datos deben ser fáciles de descubrir, acceder y reutilizar por otros equipos.
3. **Plataforma de autoservicio (Self-serve Data Platform):** Se provee una plataforma común que ofrece herramientas para publicar, consumir, procesar y asegurar datos de manera estandarizada. El objetivo es que los equipos puedan trabajar con datos sin depender de un área central de TI.
4. **Gobernanza federada (Federated Governance):** Aunque los datos se gestionan de forma descentralizada, existen políticas y estándares globales (seguridad, privacidad, formatos y calidad) que aseguran coherencia en toda la organización.

## 16. Open Data (Datos Abiertos)

### Datos Públicos y Datos Abiertos

- **Datos públicos:** datos sobre los que el público en general tiene derecho a conocer. No implica necesariamente que se pueda reutilizar o modificar.
- **Datos abiertos:** datos disponibles para cualquier persona, libres de restricciones legales o técnicas, y que pueden ser usados y redistribuidos libremente.

## Principios de los Datos Abiertos

- **Gratuitos.**
- **De libre uso:** incluso con fines comerciales (puede requerir citar la fuente).
- **Integrales:** deben incluir contexto y metadatos.
- **Oportunos:** disponibles lo suficientemente rápido para conservar valor.
- **No discriminatorios:** sin restricción de acceso.
- **Primarios:** provenientes de la fuente original, sin agregación.
- **Permanentes:** debe mantenerse el historial de versiones.
- **Legibles por máquina:** estructurados para procesamiento automático.

## Finalidades de la publicación de datos abiertos

La liberación de datos abiertos puede tener distintos objetivos principales:

- **Transparencia y participación ciudadana** (*open government*): facilitar el acceso a la información pública para fortalecer la rendición de cuentas y la participación de la sociedad civil.
- **Generación de servicios e innovación:** permitir que empresas, emprendedores e iniciativas privadas utilicen los datos para crear productos, servicios y aplicaciones.

En términos regionales:

- En **América Latina** predomina la finalidad de **transparencia y gobierno abierto**.
- En **Europa** se prioriza la reutilización de datos para fines **económicos e innovación**.

## Ley 25.326 - Protección de Datos Personales

La Ley 25.326 protege los **datos personales** almacenados en archivos, registros o bases de datos, sean públicos o privados, garantizando el derecho al honor y a la intimidad. Es relevante para el diseño y administración de sistemas de bases de datos, pues impone obligaciones sobre recolección, tratamiento, seguridad y acceso a los datos.

### Principios y Calidad de Datos

- Los datos deben ser **ciertos, adecuados, pertinentes y no excesivos** según la finalidad.
- Deben recolectarse por medios lícitos y para un fin específico, informado al titular.
- Deben mantenerse **actualizados** y eliminarse o rectificarse cuando sean incorrectos o dejen de ser necesarios.

## Consentimiento y Finalidad

- Es obligatorio el **consentimiento libre, expreso e informado** del titular, salvo excepciones (fuentes públicas, salud, obligaciones legales).
- El responsable debe informar: finalidad del tratamiento, identidad del responsable, carácter obligatorio u opcional de los datos y los derechos de acceso, rectificación y supresión.

## Seguridad y Confidencialidad

- Deben aplicarse **medidas técnicas y organizativas** para garantizar la integridad, confidencialidad y disponibilidad de los datos.
- Los responsables y usuarios de bases deben respetar el **secreto profesional**, incluso después de finalizar su relación con la base.

## Transferencia y Cesión de Datos

- Solo se pueden ceder datos con consentimiento del titular o autorización legal.
- Las transferencias internacionales solo son válidas hacia países con protección adecuada o con consentimiento explícito.
- Si un tercero procesa datos (por encargo), debe destruirlos al finalizar su tarea, salvo que exista autorización para conservarlos.

## Derechos del Titular

- Acceder a los datos (respuesta obligatoria en hasta 10 días hábiles).
- Rectificar, actualizar o suprimir datos (respuesta en hasta 5 días hábiles).
- Ejercer **habeas data** si no se cumplen estos derechos.

## Registro y Control

- Todas las bases de datos deben inscribirse ante la **Dirección Nacional de Protección de Datos Personales (DNPDP)**.
- La autoridad de control puede imponer sanciones: multas, suspensión o clausura, además de sanciones penales y civiles en caso de incumplimiento.

## Secreto Estadístico

- Protegido por leyes específicas (Ley 17.622, Decreto 3.110/70, etc.).
- La información estadística debe preservarse de forma que no identifique a personas o empresas.



Figura 41: Gauss te desea éxitos en tu examen y abrígate que hace frío.