



# Base de Datos

## ▼ I - Introducción a las Bases de Datos

### Bases de datos

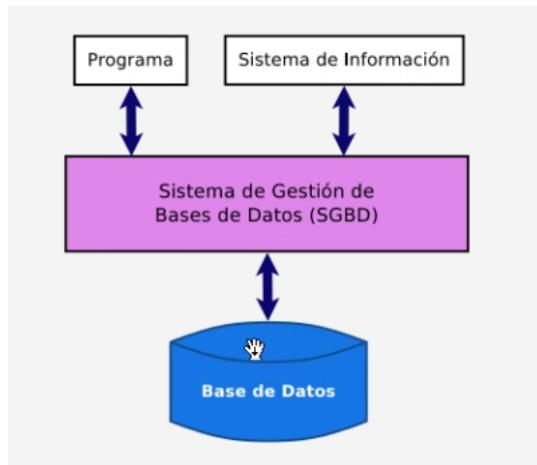
*Es un conjunto de datos interrelacionados.*

**Dato:** *Es un hecho que puede ser representado y almacenado de alguna forma, y que tiene un sentido implícito*

*Las bases de datos tradicionales almacenan proposiciones, las no tradicionales almacenan tipos de datos más complejos como imágenes, audio, video, o datos geoespaciales.*

### Sistemas de Gestión de Bases de Datos

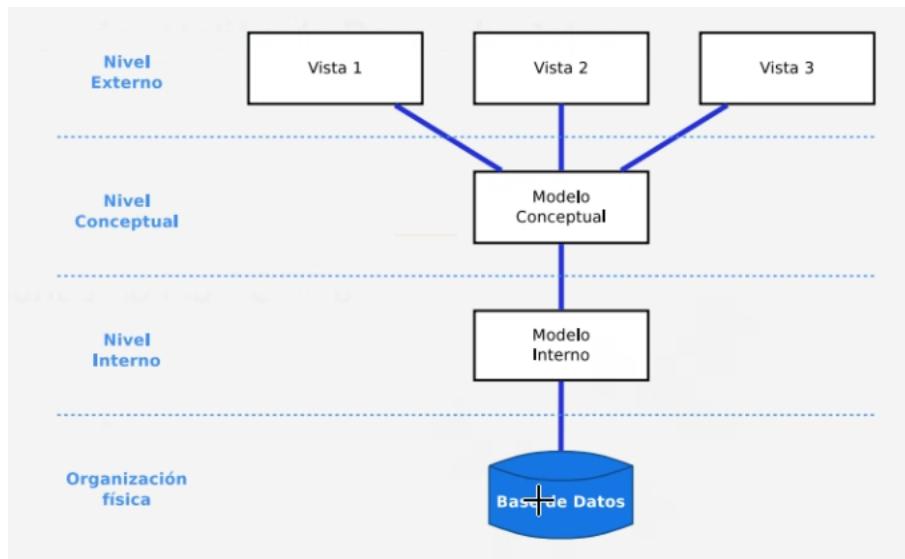
*Es un conjunto de programas que gestiona y controla la creación, manipulación y acceso a la base de datos.  
Provee un nivel de abstracción entre los programas o sistemas de información y los datos.*



Da una capa de abstracción entre el programa y la real base de datos, por ejemplo el file system de una computadora

## Arquitectura de 3 capas ANSI/SPARC

Arquitectura en 3 niveles de abstracción para la descripción/representación de los datos de una base de datos.



**Modelo Interno:** Representa la forma en que los datos se almacenan utilizando estructuras de datos y organizaciones de archivos.

**Modelo Conceptual:** Describe la semántica de los datos, abstrayéndose de su implementación física. Describe entidades, tipos de datos, operaciones y restricciones de seguridad y de integridad.

**Modelo externo:** Representa la forma en que los usuarios perciben los datos.

### Funciones de los SGBDs

- Almacenamiento y consulta
- Integridad

- Seguridad
- Conurrencia
- Recuperación
- Soporte transaccional

## Diseño Conceptual de Base de Datos: Modelo Entidad-Interrelación

**Un modelo de datos debe incluir los siguiente elementos:**

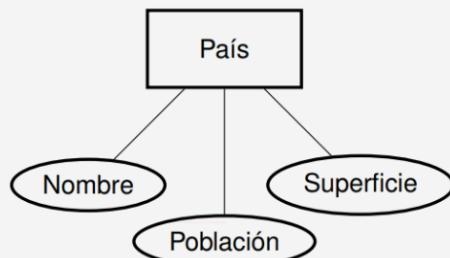
- Un conjunto de objetos, sus propiedades e interrelaciones entre ellos, que representa la estructura.
- Un conjunto de operaciones, o lenguaje, que permite manipular los datos.
- Restricciones sobre los objetos, las interrelaciones y las operaciones

## Modelo Entidad-Interrelación

Modelo esquemático basado en entidades, interrelaciones y atributos para el modelado semántico de datos.

- **Tipo de Entidad:** Es un tipo o clase de objeto en particular.
  - Ejemplos: Futbolista, país, canción
  - Una entidad es una instancia de un tipo de entidad. Ej.: Messi, Colombia
- **Atributo:** Es una propiedad que describe a la entidad.
  - Ejemplos: Futbolista → Cotización, País → Población, Canción → Duración
- **Tipo de Interrelación:** Es la definición de un conjunto de relaciones o asociaciones similares entre 2 ó más tipos de entidades.
  - Ejemplo: Futbolista nació en País
  - Una interrelación en concreto es: Didier Drogba nación en Costa de Marfil.

El diagrama Entidad-Interrelación representa los tipos de entidades y sus atributos como:



Y los tipos de interrelaciones con:

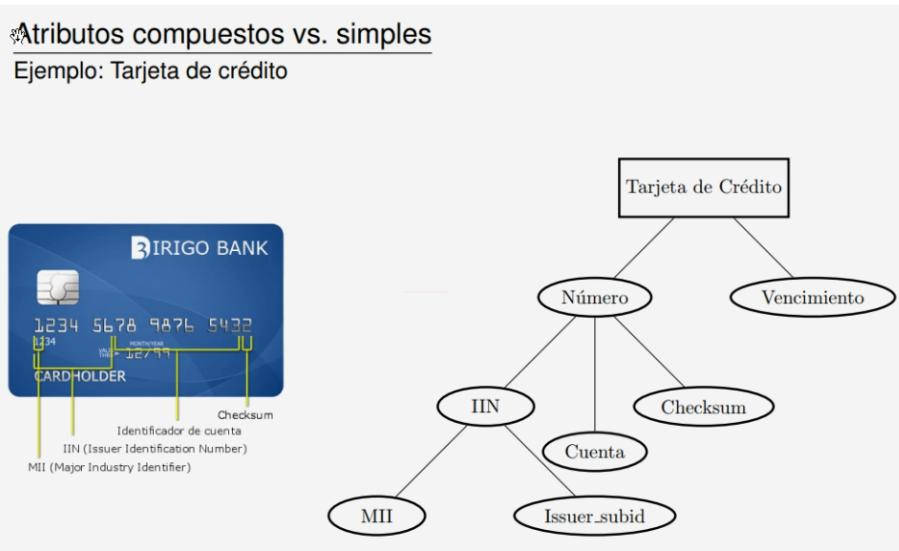


## Atributos

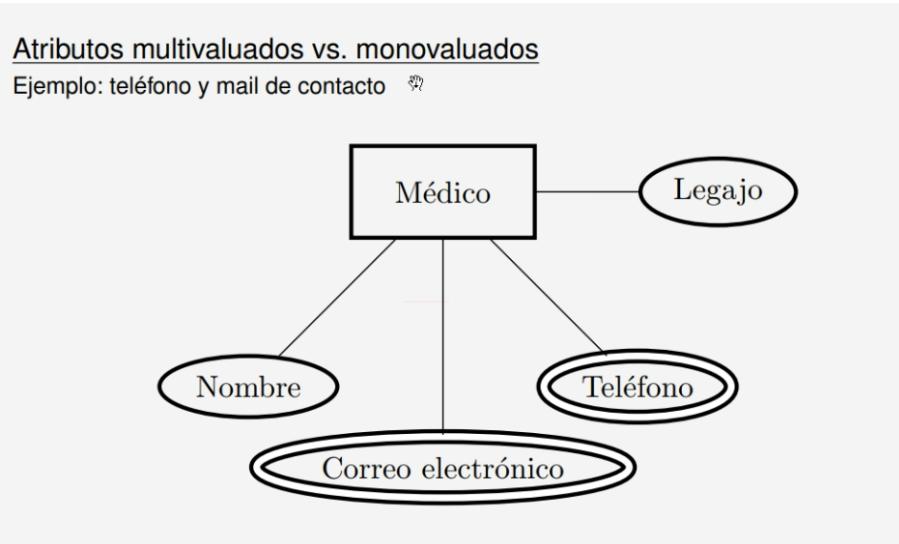
Cada entidad tendrá valores particulares para cada uno de los atributos del tipo de entidad al que corresponde.

El dominio de un atributo es el conjunto de valores que el mismo puede tomar. Por ejemplo el dominio de una cantidad es un número y no un string o un número negativo.

En ciertos casos puede permitirse que el atributo de un tipo de entidad no tome ningún valor concreto en ciertas instancias. En estos casos diremos que el atributo toma el valor nulo.



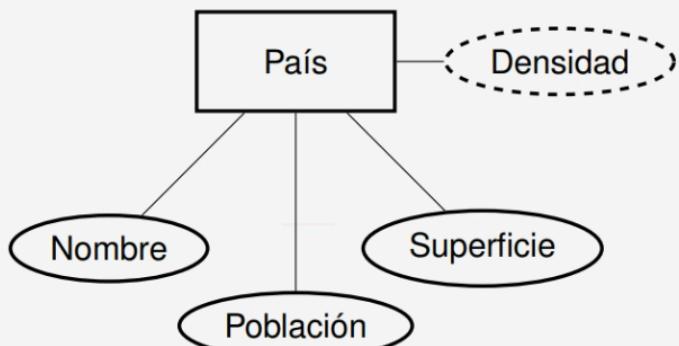
Atributos que tienen sub-atributos



Puede tener varios correos electrónicos y teléfonos.

## Atributos almacenados vs. derivados

Ejemplo: densidad de población



$$Densidad = \frac{Población}{Superficie}$$

Un atributo derivado es un atributo que puede obtenerse mediante una operación sobre los otros atributos de la entidad.

## Entidades

Al conjunto de ocurrencias o instancias de un determinado tipo de entidad en un estado determinado de la base de datos se lo denomina **conjunto de entidades** de ese tipo de entidad. Por ejemplo, para el tipo de entidad País, nuestra base de datos podría tener cargadas en un momento dado las siguientes instancias:

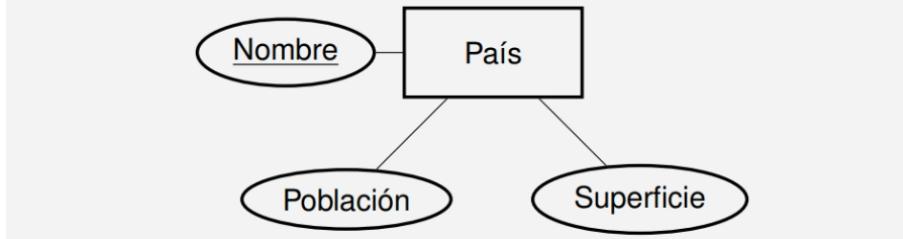
### PAÍS(nombre, población, superficie)

- (Argentina, 40.117.096, 2.780.400 km<sup>2</sup>)
- (Costa de Marfil, 22.848.945, 322.463 km<sup>2</sup>)
- (Serbia y Montenegro, 10.829.175, 102.350 km<sup>2</sup>)
- (Países Bajos, 17.000.074, 41.543 km<sup>2</sup>)

## Restricciones:

- Todo tipo de entidad debe tener un subconjunto del conjunto de atributos cuyos valores sean necesariamente distintos para cada una de las entidades en el conjunto de entidades.
- Dichos atributos se llaman **atributos clave** o **identificadores únicos**.
- Si no los encontramos, debemos crear uno (*id*).
- Al ser distintos para cada entidad, los atributos clave permiten identificar únicamente a las entidades.

En el diagrama Entidad-Interrelación los representamos subrayados:

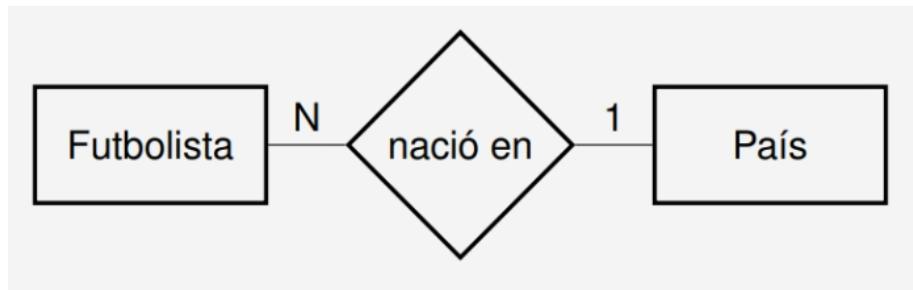


- El conjunto de atributos clave debe ser **minimal**, es decir, ningún subconjunto del mismo debe ser capaz de identificar únicamente a las entidades.
- Aún así, es posible que exista más de un conjunto de atributos clave para un tipo de entidad.

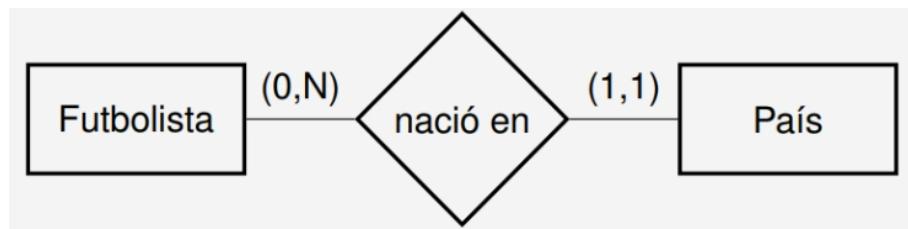
### Interrelaciones

La aridad ó grado de un tipo de interrelación es la cantidad de tipos de entidad que coparticipan del mismo. Comenzaremos analizando tipos de interrelación binarios, es decir aquellos en que participan dos tipos de entidades.

- La **cardinalidad** es la máxima cantidad de instancias de cada tipo de entidad que pueden relacionarse con una instancia concreta de los tipos de entidades restantes. Ej.:
  - Un futbolista sólo puede haber nacido en un único país.
  - En un país pueden haber nacido muchos futbolistas.

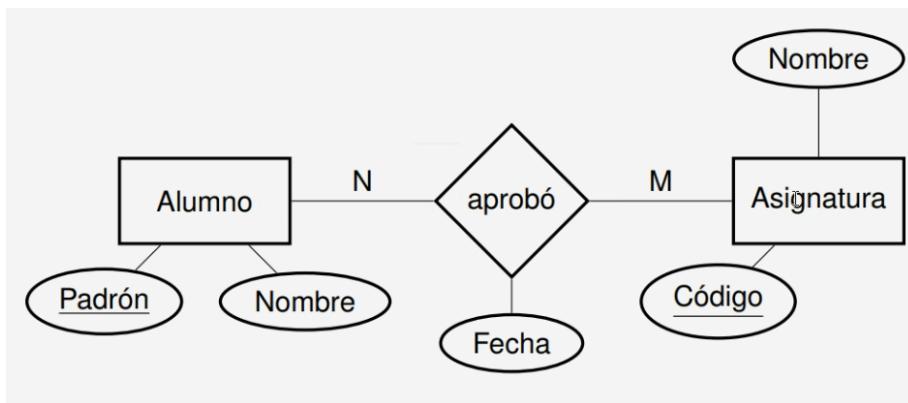


- La **participación** es la mínima cantidad de instancias de cada tipo de entidad que deben relacionarse con una instancia concreta de los tipos de entidades restantes.
- Ej.:
  - Un futbolista debe haber nacido en algún país.
  - En un país puede no haber nacido ningún futbolista.
- Cuando requerimos que cada instancia de E\_1 participe de alguna instancia de r\_1 para poder subsistir, diremos que E\_1 tiene **participación total** o **dependencia existencial** en r\_1. En caso contrario diremos que tiene **participación parcial**.
- Los indicaremos como (min, max) en el diagrama, donde min denotará la participación y max denotará la cardinalidad del tipo de entidad en una interrelación dada.

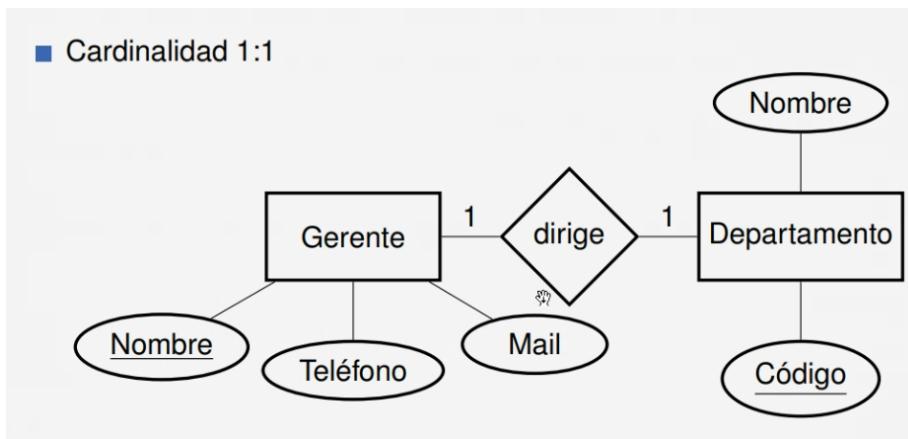


Restricciones de cardinalidad + Restricciones de participación = Restricciones estructurales

Las interrelaciones también pueden tener atributos. Ej.: registro de asignaturas aprobadas por los alumnos de una facultad.

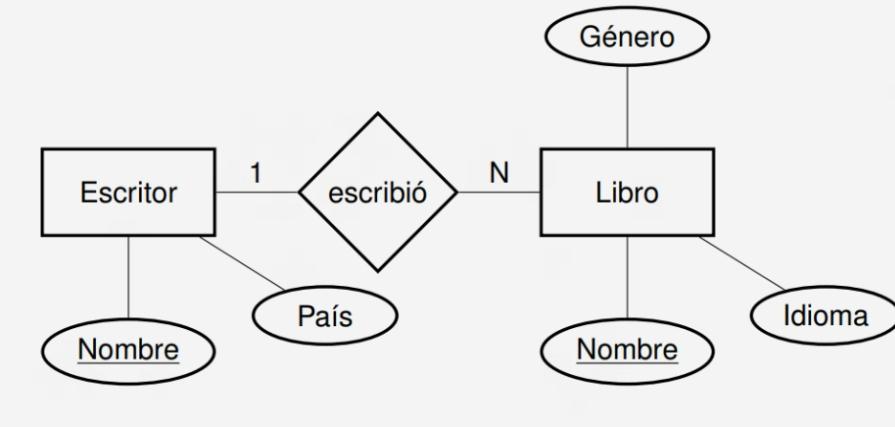


- En los tipos de interrelaciones también debemos identificar un conjunto de **atributos clave**.
- Sólo pueden formar parte de los atributos clave de una interrelación de los atributos clave de los tipos de entidad que participan de la misma.



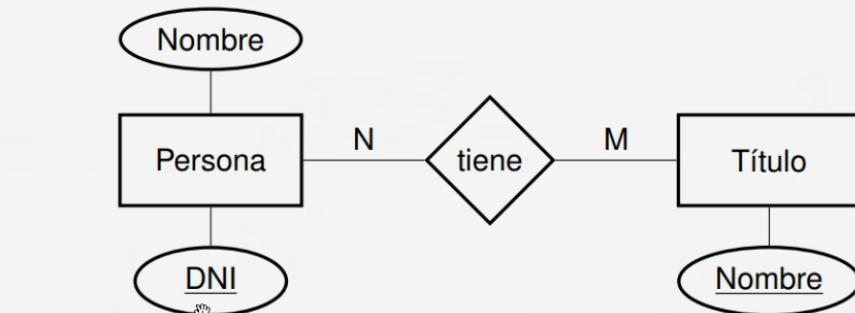
La clave es Gerente.Nombre aunque también podría ser Departamento.Código.

■ Cardinalidad 1:N



*La clave es Libro.Nombre*

■ Cardinalidad N:M



■ La clave es {Persona.DNI, Título.Nombre}

*Notar que es un conjunto con ambos elementos.*

Ejemplo 1

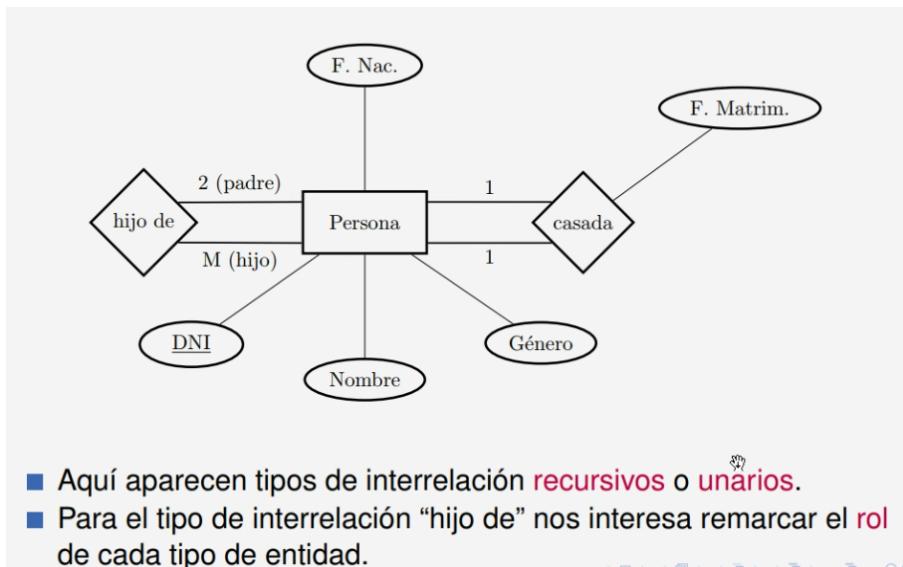
Los dueños de esta librería desean crear una base de datos que contenga información sobre los libros actualmente en venta, y que permita hacer búsquedas por nombre o país de origen del autor, género, idioma y año.

Ejemplo 2

El Registro Nacional de las Personas quiere mantener una base de datos con el nombre, DNI, género y fecha de nacimiento de cada ciudadano argentino. Asimismo desea tener registrados todos los matrimonios en curso (no divorciados) incluyendo la fecha de matrimonio, y los nacimientos de personas indicando la identidad de los padres en caso que la misma sea conocida.

Hipótesis: Suponga que todas las personas son argentinas.

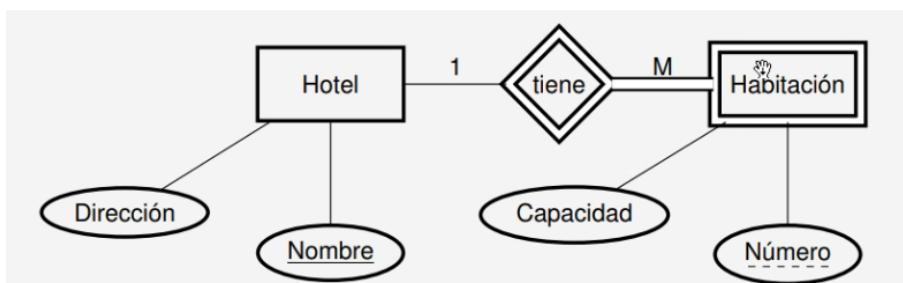
Solución:



## Modelo ER avanzado

### Entidades débiles

A veces la identificación de una identidad depende de su interrelación con otra entidad, a la cual está subordinada.



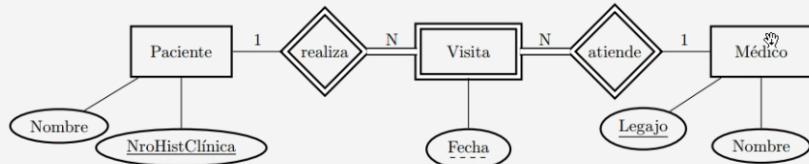
Cuando un tipo de entidad depende de otro para ser identificado, se dice que es un **tipo de entidad débil**.

La clave de una entidad débil se compone de la clave de su entidad identificadora, más algún/os atributos propios, que se denominan **discriminantes**, y se indican con líneas punteadas.

Un tipo de entidad débil siempre tiene participación total en el tipo de interrelación que la vincula con su tipo de entidad identificadora.

### Problema de paciente visita médico

- En un diseño correcto, la Visita debería ser una entidad en sí misma:

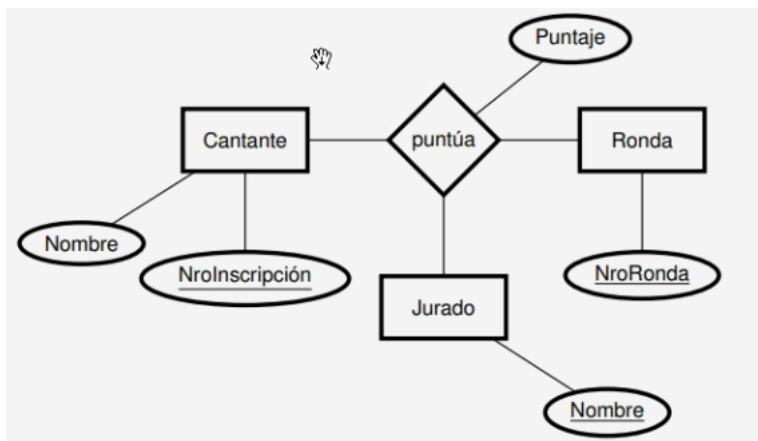


- La clave de Visita es entonces {NroHistClínica, Legajo, Fecha}.

### Interrelaciones n-arias

Son aquellas en que participan 3 tipos de entidad distintos.

Ejemplo: En un concurso de canto se organizan rondas temáticas en las que se inscriben algunos participantes. En cada ronda, los cantantes que participan son calificados por una serie de jurados.

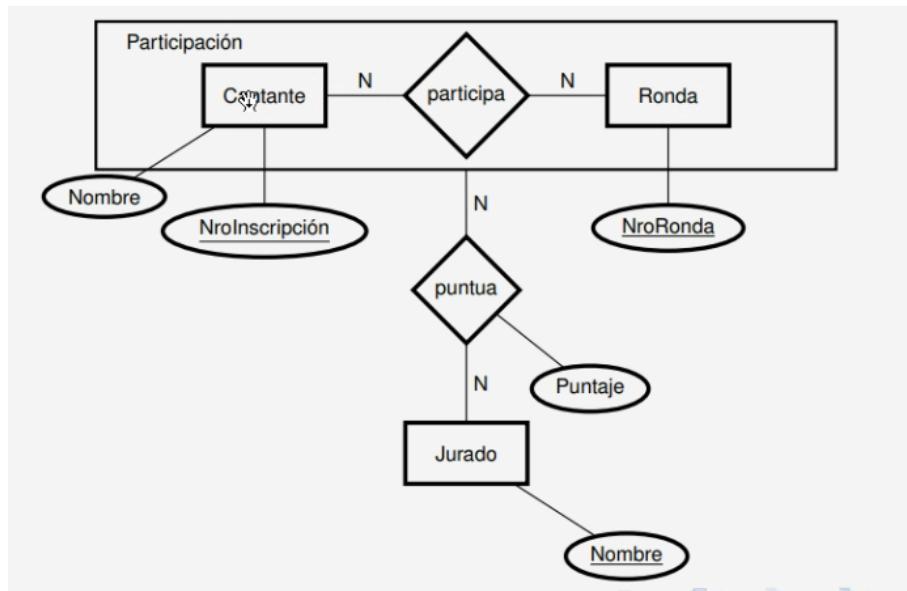


Aquí la cantidad de un tipo entidad determina la cantidad de instancias de interrelación en que puede aparecer, fijadas las instancias de los otros dos tipos de entidades. Y el conjunto de atributos clave es { Cantante.NroInscripción, Ronda.NroRonda, Jurado.Nombre }

**EL MÍNIMO DE UNA TERNARIA ES CASI SIEMPRE 0 y suele ser muy poco restrictiva, no debería ser la primera opción.**

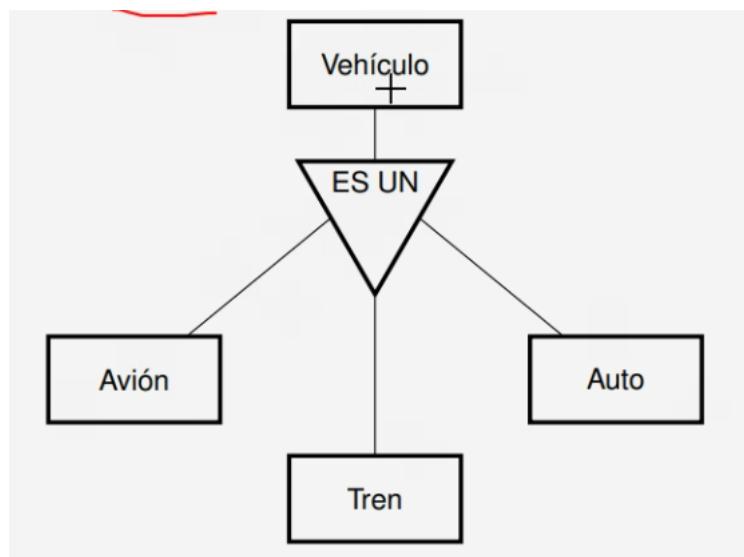
### Agregación

La desventaja del diseño anterior es que no nos permite registrar cantantes en rondas si no fueron evaluados por ningún jurado. Esto puede resolverse definiendo a la **agregación** de un cantante y una ronda como una entidad en sí misma:

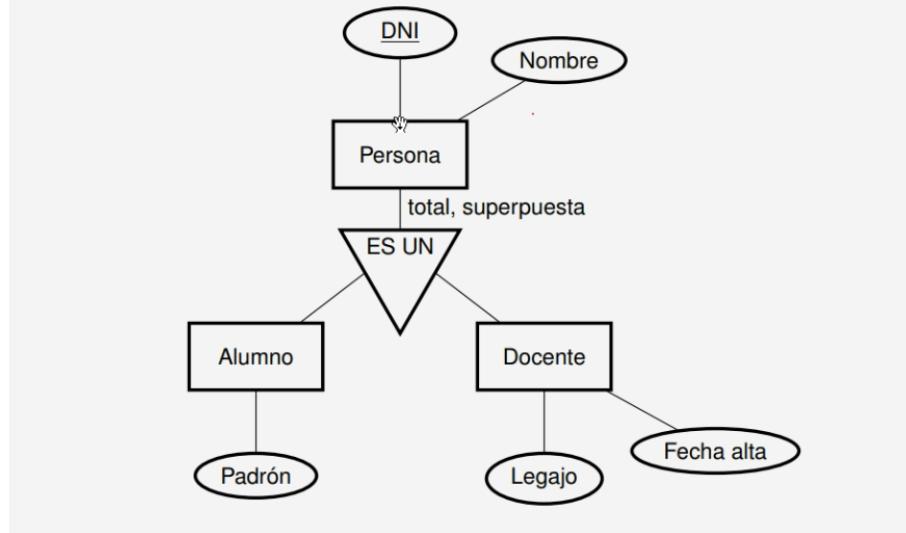


### Generalización/especialización y unión

La especialización y la generalización nos permiten representar relaciones de tipo "es un" en el modelo de datos.



**Ejemplo:** Docentes y alumnos de una facultad

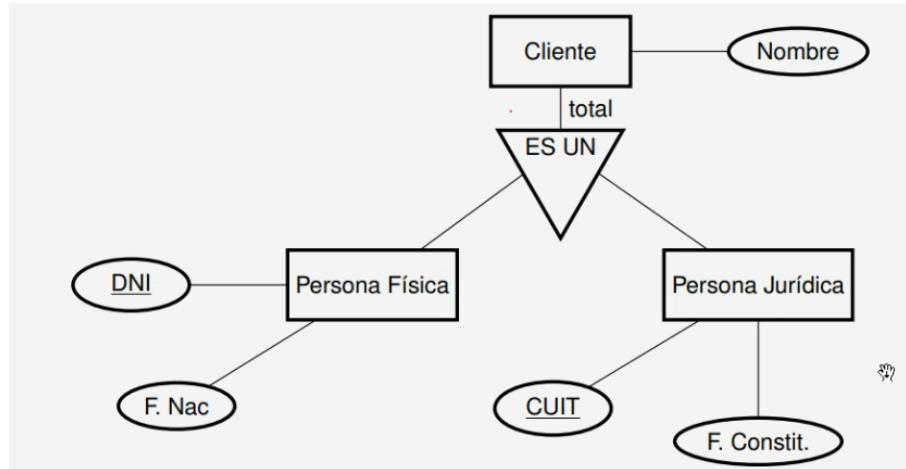


Puede ser total o superpuesta.

- **Total:** Cuando toda persona es Alumno o Docente
- **Parcial:** Cuando hay personas que no son Alumnos ni Docentes
- **Superpuesta:** Cuando hay personas que pueden ser Alumnos y Docentes a la vez
- **Disjunta:** Cuando solo hay personas que son Alumno xor Docente

## Unión

*Ejemplo: Un banco tiene como clientes tanto a personas como a personas jurídicas radicadas en Argentina.*



Un cliente ES UNA persona física o bien ES UNA persona jurídica. Pero NO necesariamente una persona física debe ser un cliente del banco.

Acá es importante que la identificación esté en el sub-tipo, pues, si estuviera en el super-tipo no podríamos saber que variante es.

## ▼ II - Modelo Relacional

### Estructura

Es una formalización matemática basada en el concepto de relación.

- **Dominio:** Es un conjunto de valores homogéneos.
  - Ejemplo:  $D_1 = \{ \text{Barcelona}, \text{Sevilla}, \text{Buenos Aires} \}$
  - $D_2 = \{ \text{Argentina}, \text{España}, \text{Chile} \}$
- **Producto cartesiano:**  $A \times B$  se define como el conjunto de pares  $(a, b)$  que cumplen que  $a \in A$  y  $b \in B$ .
  - En nuestro ejemplo,  $D_1 \times D_2 = \{ (\text{Barcelona}, \text{Argentina}), (\text{Barcelona}, \text{España}), (\text{Barcelona}, \text{Chile}), (\text{Sevilla}, \text{Argentina}) \dots \text{etc (todas las combinaciones entre los valores)} \}$
- **Relación:** Es un subconjunto de un producto cartesiano.
  - En el ejemplo anterior podría especialmente interesarnos la siguiente relación:  $R = \{ (\text{Barcelona}, \text{España}), (\text{Sevilla}, \text{España}), (\text{Buenos Aires}, \text{Argentina}) \}$

En el modelo relacional, un nombre de relación  $R$  junto con una lista de atributos asociados se denomina esquema de relación, y se denota de la siguiente manera:

- $R(A_1, A_2, \dots, A_n)$
- Ejemplo: Películas(*nombre\_pelicula*, *año*, *nombre\_director*, *cant\_oscars*)

Cada uno de los atributos  $A_i$  de un esquema de relación está asociado a un dominio particular,  $\text{dom}(A_i)$ . En nuestro ejemplo:

- $\text{nombre_pelicula} \rightarrow \text{dom(nombre_pelicula)} = \text{string}$
- $\text{año} \rightarrow \text{dom(año)} = \text{naturales positivos}$
- $\text{nombre_director} \rightarrow \text{dom(nombre_director)} = \text{string}$
- $\text{cant_oscars} \rightarrow \text{dom(cant_oscars)} = \text{naturales positivos}$

Un elemento en una relación se denomina tupla.

- $t = (\text{Kill Bill}, 2003, \text{Quentin Tarantino}, 0)$  es una tupla de Películas.
- Lo denotamos como  $t$  in Películas
- $u = (\text{Kill Bill}, 2003, \text{Quentin Tarantino}, 3)$  no es una tupla de Películas.
- $u$  not in Películas

El valor tomado por un atributo  $A$  en la tupla  $t$  se denota  $t[A]$  o  $t.A$ .

- $t[\text{nombre_director}] = t.\text{nombre_director} = \text{Quentin Tarantino}$ .
- $t[\text{nombre_pelicula}, \text{anio}] = t.(\text{nombre_pelicula}, \text{anio}) = (\text{Kill Bill}, 2003)$ .

La cardinalidad de una relación  $R$  es la cantidad de tuplas que posee.

- La simbolizaremos  $n(R)$
- Ejemplo:  $n(\text{Películas}) = 4$

- Una forma útil de representar una relación es a través de una **tabla** en la que las **columnas** representan los atributos y las **filas** representan las tuplas. En el caso de la relación Películas:

nombre_pelicula	año	nombre_director	cant_oscars
Kill Bill	2003	Quentin Tarantino	0
Django Unchained	2012	Quentin Tarantino	2
Star Wars III	2005	George Lucas	0
El Cisne Negro	2010	Darren Aronofsky	1

## Restricciones

- Las relaciones del modelo relacional representan generalmente entidades o interrelaciones de nuestro modelo de datos.
- Deben cumplir una serie de restricciones de distintos tipos.
- Los atributos deben ser atómicos (no se permiten atributos compuestos o multivaluados)
- No pueden existir dos tuplas distintas que coincidan en los valores de todos sus atributos (no puede estar más de una vez una tupla)
- Sin embargo, generalmente existe un subconjunto SK del conjunto de atributos (A1, A2, ..., An) de R que cumple con la condición de que dadas dos tuplas s, t in R, las mismas difieren en al menos uno de los atributos de Sk.
- Cuando un subconjunto SK cumple esta propiedad, diremos que SK es una superclave de R.
- Nos interesan aquellas superclaves que son minimales, es decir que no admiten ningún subconjunto propio con la misma propiedad. A estas superclaves las llamaremos claves candidatas o simplemente claves.
- De entre todas las claves candidatas elegiremos una como clave primaria de la relación, La indicaremos subrayada en el esquema.

**La superclave permite identificar a una tupla de otras, la clave candidata es el subconjunto de superclaves minimales y clave primaria es alguna de las candidatas.**

Las bases de datos almacenan múltiples esquemas de relación, muchas veces relacionados entre ellos. En el modelo relaciones, una base de datos se representa a través de un esquema de base de datos relacional. Un esquema de base de datos relaciones S es un conjunto de esquemas de relación  $S = \{R_1, R_2, \dots, R_n\}$  junto con una serie de restricciones de integridad.

Ejemplo: Cine = {Películas, Actores, Actuaciones}, en donde:

- Películas(nombre\_pelicula, año, nombre\_director, cant\_oscars)
- Actores(nombre\_actor, país)
- Actuaciones(nombre\_pelicula, nomber\_actor)
- Restricción de integridad de entidad: La clave primaria de una relación no puede tomar el valor nulo.
- Restricción de integridad referencial: Cuando un conjunto de atributos FK de una relación R hace referencia a la clave primaria de otra relación S, entonces para toda tupla de R debe existir una tupla de S cuya clave primaria sea igual el valor de FK, a menos que todos los atributos de FK sean nulos.

Cine = {Películas, Actores, Actuaciones}

- Películas(nombre\_película, año, nombre\_director, cant\_oscars)
- Actores(nombre\_actor, país)
- Actuaciones(nombre\_pelicula, nombre\_actor)

- Ejemplo: Si una tupla en Actuaciones hace referencia "Star Wars III", entonces debe existir "Star Wars III" en la relación Películas.

### Resumen

- **Dominio:** Si la columna es de tipo carácter, no le puedo poner un número
- **Unicidad:** No pueden haber dos tuplas con el mismo valor en la clave primaria
- **Integridad de entidad:** No se puede desconocer ninguno de los atributos que conformen la clave primaria de una relación
- **Integridad referencial:** En una clave foránea puede existir un campo con los demás campos nulos pero de no ser así, la referencia tiene que ser válida, no podemos tener punteros a datos que no existen

Cine = {Películas, Actores, Actuaciones}

Películas(nombre\_pelicula, año, nombre\_director, cant\_oscars)

Actores(nombre\_actor, país)

Actuaciones(nombre\_pelicula, nombre\_actor)

- En nuestro ejemplo de la base de datos Cine, Actuaciones.nombre\_pelicula es clave foránea y hace referencia a la relación Películas. Asimismo, Actuaciones.nombre\_actor es clave foránea y hace referencia a la relación Actores.
- Indicaremos a las claves foráneas con un subrayado punteado.

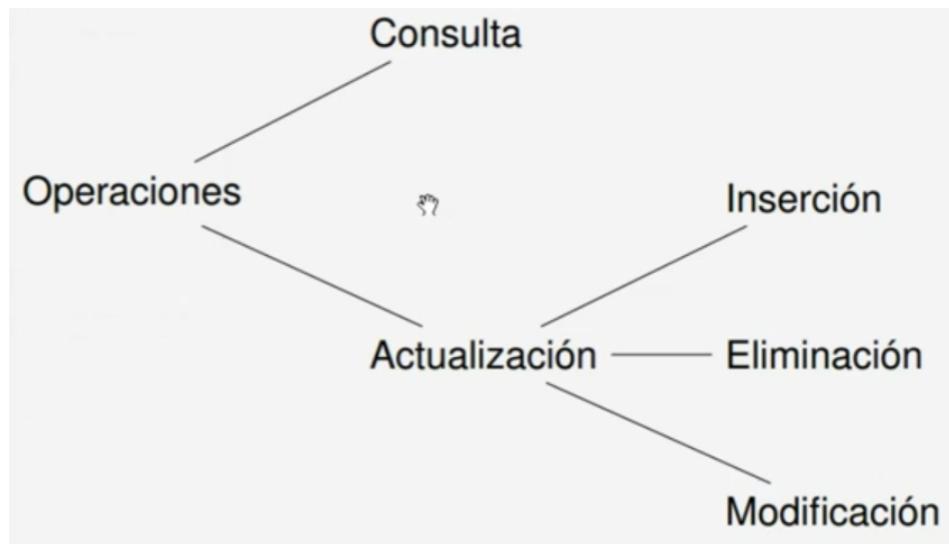
En este caso nombre\_pelicula y nombre\_actor son clave primaria de Actuaciones, entonces:

FK: {nombre\_pelicula}, {nombre\_actor}

PK: {nombre\_pelicula, nombre\_actor}

### Operaciones

Las operaciones del modelo relacional se especifican a través de lenguajes como el álgebra relacional o el cálculo relacional.



*En consultas no se deben chequear ninguna de las restricciones*

*En inserciones se deben chequear las 4 restricciones*

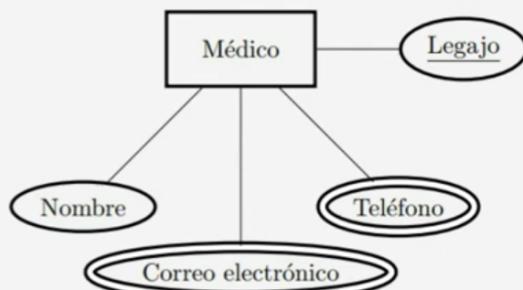
*En eliminaciones se debe chequear Integridad Referencial, hay tres alternativas:*

- Rechazar la eliminación
- Eliminar en cascada
- Poner en NULL los atributos referenciales de las tuplas de R

*En modificación se deben chequear Dominio, Unicidad si se cambia un atributo de clave primaria, Integridad de entidad si el nuevo valor es NULL y forma parte de clave primaria, Integridad Referencial tomando las 3 alternativas de arriba.*

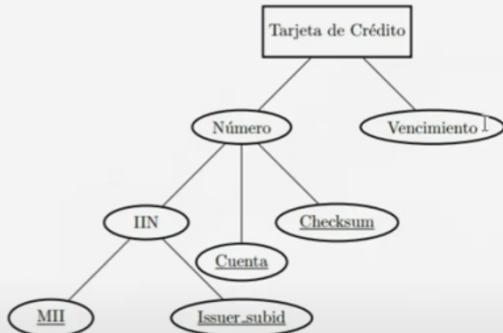
**Transaccion:** A veces es necesario realizar una serie de operaciones por completo, o bien no realizarlas, surge el concepto de transaccion, como conjunto ordenado de operaciones que, o se ejecutan por completo, o no se ejecutan. La ejecución de una transaccion es a todo o nada. Si una transaccion no puede terminar de realizarse porque una de sus operaciones viola alguna restricción de integridad, entonces debe dejarse la base de datos en el estado anterior al inicio de la misma.

#### ■ Atributos multivaluados:



<i>Médicos(legajo_médico, nombre_médico)</i> <i>Telefonos(legajo_médico, teléfono)</i> <i>Mails(legajo_médico, mail)</i>
--

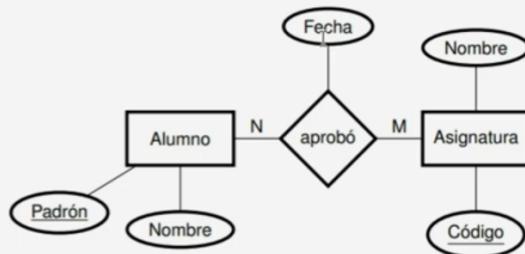
■ Atributos compuestos:



TarjetasCrédito(MII, issuer\_subid, cuenta, checksum, fecha\_venc)

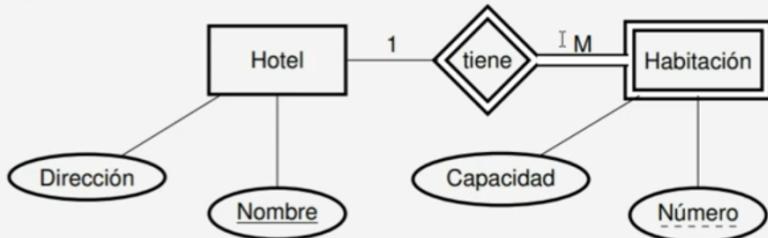
■ Se representan a través de sus sub-atributos simples.

■ Cada interrelación N:M del modelo ER producirá una relación del modelo relacional.



Alumnos(padrón, nombre\_alumno)  
 Asignaturas(código\_asignatura, nombre\_asignatura)  
 Aprobaciones(padrón, código\_asignatura, fecha\_aprobación)

■ Entidades débiles:



Hoteles(nombre\_hotel, dirección)  
 Habitaciones(número\_habitación, nombre\_hotel, capacidad)

Nota: No tiene sentido agregar una relación que represente la interrelación "tiene".

▼ III - Álgebra Relacional

Es un lenguaje del modelo relacional, es procedural.

Una operación es una función cuyos operandos son una o mas relaciones, y cuyo resultado es también una relación.

O:  $R_1 \times R_2 \times \dots \times R_n \rightarrow S$

La aridad es la cantidad de operandos que toma una operación.

## Operaciones

El operador selección (*sigma*) es un operador unario.

Dada la relación R(A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>) y la condición que se aplica a cada tupla de R,  $\sigma_{\text{condición}}(R)$  selecciona aquellas tuplas de R para las cuales la condición es la verdadera.

NationalTeams			
short_name	name	group	continent
ARG	Argentina	C	5
AUS	Australia	D	2
BEL	Belgium	F	3
BRA	Brazil	G	5
CMR	Cameroon	G	1

$\downarrow \sigma_{\text{group}='G'}(\text{NationalTeams})$

short_name	name	group	continent
BRA	Brazil	G	5
CMR	Cameroon	G	1

El operador proyección (*pi*) es también un operador unario.

Dada la relación R(A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>) y una lista de atributos L = (L<sub>1</sub>, L<sub>2</sub>, ..., L<sub>k</sub>), con L<sub>i</sub> in (A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>),  $\pi_L(R)$  devuelve una relación cuyas tuplas representan los posibles valores de los atributos de L en R.

Podemos pensar que lo que hace es proyectar cada tupla de R a un espacio de menor dimensión en que solo se conservan los atributos que están en L.

**ArgPlayers**

id	name	local_club
551	Rodrigo De Paul	Atlético Madrid
615	Thiago Almada	Atlanta Utd
674	Ángel Correa	Atlético Madrid
675	Ángel Di María	Juventus

$\pi_{local\_club}(ArgPlayers)$

**local\_club**

Atlético Madrid
Atlanta Utd
Juventus

*El operador asignación ( $\leftarrow$ ) es también un operador binario.*

Sirve para guardar en una variable el resultado de una consulta parcial.

- ¿Cómo listamos los nombres de los países del grupo G?

**NationalTeams**

short_name	name	group	continent
AUS	Australia	D	2
BEL	Belgium	F	3
BRA	Brazil	G	5
CMR	Cameroon	G	1
...	...	...	...

$Temp \leftarrow \sigma_{group='G'}(NationalTeams)$

$Selecciones\_GrupoG \leftarrow \pi_{name}(Temp)$

- Podemos también hacerlo en un único paso:

$Selecciones\_GrupoG \leftarrow \pi_{name}(\sigma_{group='G'}(NationalTeams))$

**Selecciones\_GrupoG**

name
Brazil
Cameroon
...

*El operador redenominación (ro) permite modificar los nombres de las tablas*

**ArgPlayers**

name	local_club
Rodrigo De Paul	Atlético Madrid
Thiago Almada	Atlanta Utd
Ángel Correa	Atlético Madrid
Ángel Di María	Juventus

$\rho_{Argentinos(nombre,club\_local)}(ArgPlayers)$

**Argentinos**

nombre	club_local
Rodrigo De Paul	Atlético Madrid
Thiago Almada	Atlanta Utd
Ángel Correa	Atlético Madrid
Ángel Di María	Juventus

## Operadores binarios

*Unión: Dadas dos relaciones  $R(A_1, A_2, \dots, A_n)$  y  $S(B_1, B_2, \dots, B_n)$ , la unión  $R \cup S$  es una relación que contiene a todas las tuplas de  $R$  y de  $S$ . Además para calcular la unión entre  $R$  y  $S$  las relaciones deben coincidir en sus atributos en lo que respecta al dominio. Es decir,  $\text{dom}(A_i) = \text{dom}(B_i)$ . Esta condición se denomina compatibilidad de union o compatibilidad de tipo.*

*Intersección: Es lo mismo pero conserva las tuplas que se encuentran presentes tanto en  $R$  como en  $S$ .*

*Diferencia: Es lo mismo pero conserva solo aquellas tuplas de  $R$  que no pertenecen a  $S$ .*

*Producto Cartesiano: Dadas dos relaciones  $R(A_1, A_2, \dots, A_n)$  y  $S(B_1, B_2, \dots, B_n)$ , el producto cartesiano  $R \times S$  produce una nueva relación  $T$  cuyas tuplas son todas aquellas de la forma  $(t_1, t_2, \dots, t_n, t_{n+1}, t_{n+2}, \dots, t_{n+m})$ , con  $(t_1, t_2, \dots, t_n)$  in  $R$  y  $(t_{n+1}, t_{n+2}, \dots, t_{n+m})$  in  $S$ .*

*Junta: Es un producto cartesiano seguido de una selección*

*División: Útil para comparar cuando algún elemento se vincula con todos los elementos de otro conjunto.*

## División

- Esta vez, primero el ejemplo...
- Nos interesa saber qué alumnos aprobaron los 3 TPs.

### NOTAS

alumno	TP	nota
Pedro	1	7
Pedro	3	2
Juan	1	3
Juan	2	6
Juan	3	8
Walter	1	4
Walter	2	9
Walter	3	8



### APROBADOS

alumno	TP
Pedro	1
Juan	2
Juan	3
Walter	1
Walter	2
Walter	3

REQUISITOS
TP
1
2
3

*Junta Externa: Es un left join*

## Junta externa

- Supongamos que quisieramos juntar la tabla de Scores con la de Matches, pero sin que desaparezca ningún partido.

Matches			
id	home	away	datetime
5	DEN	TUN	2022-11-22 10:00:00
6	MEX	POL	2022-11-22 13:00:00
7	FRA	AUS	2022-11-22 16:00:00

Scores					
id	match_id	team_id	player_id	minute	score_type
17	7	FRA	16	27	1
18	7	FRA	502	32	1
19	7	FRA	377	68	1
20	7	FRA	502	71	1
21	7	AUS	132	9	1

↓ (Matches  $\bowtie$   $Matches.id=match\_id$  Scores)

Matches.id	home	away	datetime	Scores.id	match_id	team_id	player_id	minute	score_type
5	DEN	TUN	2022-11-22...						
6	MEX	POL	2022-11-22...						
7	FRA	AUS	2022-11-22...	17	7	FRA	16	27	1
7	FRA	AUS	2022-11-22...	18	7	FRA	502	32	1
7	FRA	AUS	2022-11-22...	19	7	FRA	377	68	1
7	FRA	AUS	2022-11-22...	20	7	FRA	502	71	1
7	FRA	AUS	2022-11-22...	21	7	AUS	132	9	1

- El resultado muestra cada gol junto con el partido en que ocurrió, mostrando también los partidos en que no hubo goles.

## ▼ IV - SQL I

### Esquemas

*Para crear esquemas se usa el comando CREATE SCHEMA*

```
CREATE SCHEMA nombre_esquema [ AUTHORIZATION AuthId ];  
CREATE SCHEMA empresa [ AUTHORIZATION lroman ];
```

## Tablas

Para definir estructuras de tablas se usa el comando *CREATE TABLE*

```
CREATE TABLE nombre_tabla (  
    definicion_columna_1 [,  
    definicion_columna_2,  
    ...,  
    definicion_columna_N ] [,  
    definicion_restriccion_1,  
    ....,  
    definicion_restriccion_M ]  
    -- No es obligatorio definir restricciones, pero  
    -- MUY recomendado siempre definir una de PK  
);
```

La definición de columnas tiene el siguiente formato

```
nombre-columna tipo_dato [ restricciones ]  
  
-- Si no pasa el check rechaza la insercion/modificacion del campo  
padron INTEGER NOT NULL CHECK (padron > 10000)
```

## Tipos de Datos

```
INTEGER | INT, SMALLINT, FLOAT(n), DOUBLE PRECISION, NUMERIC(precision, escala)  
CHARACTER(n) | CHAR(n), CHARACTER VARYING(n) | VARCHAR(n)  
DATE, TIME, TIMESTAMP, INTERVAL  
BOOLEAN, CLOB, BLOB, UUID
```

## Restricciones

```
nombre_columna tipo_dato [ restricciones ]
```

```
PRIMARY KEY, FOREIGN KEY, UNIQUE, NULL, NOT NULL, DEFAULT valor_defecto, CHECK condicion
```

Las restricciones pueden definirse luego de las columnas y usando *CONSTRAINT* se les puede dar un nombre

```
CONSTRAINT cc_alumno_cuit UNIQUE (cuit)
```

```
-- Ejemplo  
CREATE TABLE Persona (  
    dni INT PRIMARY KEY,  
    nombre VARCHAR(255) NOT NULL,  
    fecha_nacimiento DATE  
);
```

```

CREATE TABLE HijoDe (
    dni_hijo INT,
    dni_padre INT,
    PRIMARY KEY (dni_hijo, dni_padre),
    FOREIGN KEY (dni_hijo) REFERENCES Persona(dni),
    FOREIGN KEY (dni_padre) REFERENCES Persona(dni)
);

```

## Restricción de clave foránea

*En una restricción de CLAVE FORANEA podemos opcionalmente definir estrategias en DELETE o UPDATE de la tabla referenciada*

```

[ CONSTRAINT nombre_restriccion ]
FOREIGN KEY (columnas)
REFERENCES tabla_referenciada (columnas_referenciadas)
[ ON DELETE SET NULL | RESTRICT | CASCADE | SET DEFAULT ]

```

## Claves sustitutas

- *Muy dependientes de cada motor*
- *PostgreSQL tiene los tipos SERIAL y BIGSERIAL de 4 y 8 bytes respectivamente*
- *SQL Server tiene el tipo IDENTITY*
- *MySQL tiene la opción AUTO\_INCREMENT*
- *Oracle usa secuencias y triggers BEFORE INSERT*
- *Todo esto puede cambiar en versiones posteriores, consultar siempre la documentación*

## Cambios en Tablas

*Una vez creadas las tablas pueden cambiar su estructura usando ALTER TABLE*

```

ALTER TABLE Persona ADD COLUMN direccion VARCHAR(255);

ALTER TABLE Persona RENAME TO DatosPersonas;

```

*Y para quitar tablas se usa DROP TABLE*

```
DROP TABLE DatosPersonas [ CASCADE ];
```

## Índices

*Los índices se pueden utilizar para mejorar la performance de acceso a los datos. Aunque su mantenimiento tiene un costo, con lo que pueden impactar negativamente en la performance*

```

CREATE [ UNIQUE ] INDEX nombre_indice
    ON nombre_tabla (expresion1 [, ..., expresionN] )

```

```
[ INCLUDE (columna1 [, ..., columnaN] ) ]  
[ WHERE condicion ];
```

```
-- Ejemplo  
CREATE INDEX idx_padron ON alumnos (padron);  
  
-- Para quitar indices tambien se utiliza DROP  
DROP INDEX nombre_indice;
```

Pueden incluirse columnas no indexadas en el índice

```
CREATE INDEX idx_padron ON alumnos (padron)  
INCLUDE (nombre, apellido);
```

Esto permite buscar por *padrón* y devolver *nombre* y/o *apellido* sin acceder a la tabla (mas rápido), no agiliza búsquedas por *nombre* ni *apellido*. Pero el índice ocupa mas que si no tuviera las columnas. Usar con cuidado, no incluir columnas muy grandes ni muchas columnas "porque sí".

## Índices Parciales

Puede no indexar toda la tabla

```
CREATE INDEX idx_clientes ON clientes (id)  
WHERE anulado = FALSE;
```

Solo se indexan las filas que cumpla la condición. Índice mas pequeño suele tener mejor performance. Solo se usara el índice para consultas que tengan incluida esa condición. Consultar clientes anulados no aprovechará el índice.

## SELECT

El comando para efectuar consultas en SQL es **SELECT**:

```
SELECT [ DISTINCT ] columna1 [, ..., columnaN]  
FROM tabla1 [, ..., tablaN]  
[ WHERE condicion ];
```

Es equivalente\* a un producto cartesiano, selección y proyección

$\Pi_{\text{columna1}, \dots, \text{columnaN}} (\sigma_{\text{condicion}} (\text{tabla1} \times \dots \times \text{tablaN}))$

\*si se usa *DISTINCT*

## Espacio y Comentarios

No hay problemas en poner varios espacios, tabs o incluso saltos de líneas en el comando. Un comando SQL finaliza con un ; pero no es obligatorio si se ejecuta un único comando en la consulta. Se pueden poner comentarios de línea con dos guiones

```
-- Comentario que ayude a la memoria
```

## DISTINCT

El DISTINCT hace que no se devuelvan filas repetidas. Por defecto, se comporta distinto al álgebra relacional y aplica a la fila completa, no se puede hacer por solo una de las columnas devueltas

## Expresiones

Se permite devolver una lista de expresiones, que pueden ser:

- Nombres de columnas: se devuelve el valor para cada fila. Se puede usar nombre\_tabla.nombre\_columna ante ambigüedad
- Valores constantes
- Funciones con o sin parámetros, que son a su vez expresiones
  - Un caso especial son las funciones de agregación
- Operadores aplicados a expresiones

Si se pone un asterisco (SELECT \*) se devuelven todas las columnas de la/s tabla/s

## Operadores

Para textos tenemos el operador de concatenación ||

```
apellido || ', ' || nombre -- Román, Lucas
```

Para valores numéricos los operadores matemáticos +-\* /

- Las fechas pueden restarse entre si para obtener diferencias de días, o se les puede sumar días como enteros

```
vampo_fecha + 7 -- Sumar una semana
```

Están los operadores lógicos AND, OR, y NOT

Y los de comparación: =, <>, >, ≥, < y ≤

## Rangos

Para buscar que un valor este entre dos rangos esta el operador BETWEEN

```
expresion BETWEEN rango1 AND rango2  
-- Es equivalente a  
expresion >= rango1 AND expresion <= rango2
```

## Comparación con nulos

*El nulo se toma como desconocido, entonces no se puede comparar con operadores normales*

```
expresion = NULL -- Siempre devuelve falso  
expresion <> NULL -- Tambien siempre devuelve falso
```

*Se debe usar el operador IS (o IS NOT para el contrario)*

```
expresion IS NULL -- Verdadero si expresion es nula  
expresion IS NOT NULL -- Verdadero si no es nula
```

## Funciones de conversión de datos

*Existen distintas funciones de conversión: to\_char, to\_date, to\_number, to\_timestamp*

```
to_char(padron, 'fm000000') -- 6 digitos, 0s a la izq
```

*Se puede extraer un subcampo de una fecha*

```
EXTRACT (DAY FROM current_date)
```

```
DAY(current_date) -- En MS SQL, no standard
```

## Cast

*Si bien no es una función, se puede castear entre tipos de dato*

```
CAST(codigo AS CHAR(2)) -- no agrega 0s a la izq
```

## Otras funciones

*Numéricas: ABS, ROUND, TRUNC, LN, LOG, EXP, SQRT, etc*

```
round(0.345, 2) -- 0.35
```

*Strings: CHARACTER\_LENGTH, TRIM, REPLACE, SUBSTR, LOWER, UPPER*

```
upper('HaCkEr') -- HACKER
```

*De tiempo: CURRENT\_TIME, CURRENT\_DATE*

## Comparación de Patrones

*Para comparar un string con patrones se utiliza LIKE*

- % se usa para cualquier cadena de 0 o mas caracteres
- \_ para un caracter cualquiera

```
nombre LIKE 'A%o' -- Alejandro, Arnaldo, etc...
```

*Si no queremos que cumpla un patrón, usamos NOT LIKE*

*Otros operadores de patrones:*

- *ILIKE*: Similar al *LIKE* pero case-insensitive (no es SQL standard)
- *SIMILAR TO*: Mas parecido a expresiones regulares

## Expresiones condicionales

Con la expresión CASE podemos hacer que el valor dependa de condiciones

```
CASE WHEN condicion1 THEN valor1
      [ WHEN condicion2 THEN valor2 (...) ]
      [ ELSE valor_por_defecto ]
END

CASE WHEN intercambio = TRUE THEN 'De intercambio'
      WHEN intercambio = FALSE THEN 'Propio'
      ELSE 'Indeterminado'
END AS "tipo de estudiante"
```

## Condición del WHERE

Es una expresión de que debe ser de tipo lógico:

```
WHERE monto < (total / 2) -- menos de la mitad
```

Se devolverán las filas en las cuales al evaluar la condición, el resultado devuelva el valor VERDADERO. No quiere decir que se evalúe la condición fila a fila, el procedimiento puede ser otro, pero con un resultado equivalente. Si no se pone condición, se devuelven todas las filas.

## Alias

Se le pueden poner un alias a las columnas devueltas

```
apellido || ', ' || nombre AS "Ape y Nom"

-- Y tambien a las tablas
FROM tabla [ AS ] t WHERE t.campo = 1

-- Es necesario usarlo cuando se usa mas de una vez una tabla.
-- Tambien se pueden renombrar sus columnas
FROM tabla AS t1(a1, b1), tabla as t2(a2, b2)
```

## Ordenamiento

Se puede ordenar el resultado de una consulta por una o mas expresiones. Se ordena por la primera, y si hay empates por la segunda, etc. Se puede definir ordenes ascendentes o descendentes de cada expresión

```
ORDER BY expr1 [ ASC | DESC ]
[, (...), exprN [ ASC | DESC ]]
```

Si se usan números como expresiones, se ordena por las columnas devueltas en el SELCET (numeradas desde 1)

## Limitar filas devueltas

Se puede solicitar que no se devuelvan mas de *N* filas

```
FETCH FIRST cantidad_filas ROWS ONLY
```

Distintos motores tienen distintas formas de hacerlo

- *LIMIT cantidad\_filas*
- *SELECT TOP cantidad\_filas (...)*

Debería venir siempre después de un *ORDER BY* o ejecuciones diferentes podrían tener distintos resultados

## Paginado

Se pueden saltar *M FILAS* PREVIO A DEVOLVER LAS *n FILAS*

```
OFFSET filas_saltear ROWS  
FETCH FIRST cantidad_filas ROWS ONLY
```

Esto permite manejar paginado de los datos

## Join

Para hacer juntas se puede utilizar el operador *JOIN*

```
FROM tabla1 INNER JOIN tabla2 ON condicion_de_join
```

Si ambas tablas tienen columnas de un mismo nombre y se va a comparar por igualdad, se puede usar *USING*

```
FROM tabla1 INNER JOIN tabla2 USING (columna1 [, columna2, ..., columnaN])
```

también esta el *NATURAL JOIN*, con su condición implícita

```
FROM tabla1 NATURAL JOIN tabla2
```

## Joins Externos

El *Join externo* devuelve siempre todas las filas de alguna de las tablas, aun cuando no se vincule con ninguna fila de la otra. En ese caso, los valores de las columnas de la otra tabla se devolverán siempre como nulos

```
FROM tabla1 LEFT OUTER JOIN tabla2 ON condicion  
FROM tabla1 RIGHT OUTER JOIN tabla2 ON condicion  
FROM tabla1 FULL OUTER JOIN tabla2 ON condicion
```

## Operadores de conjuntos

Los tres operadores de conjunto están presentes en SQL

```
SELECT ... UNION [ ALL ] SELECT ...;
SELECT ... INTERSECT [ ALL ] SELECT ...;
SELECT ... EXCEPT [ ALL ] SELECT ...;
```

Las consultas deben tener misma cantidad y tipo de datos de columnas (unión compatible). La opción ALL hace que la respuesta sea un set (igual que álgebra relacional). Si no, es un multiset (con duplicados).

## Consultas anidadas

El resultado de una consulta SQL es una tabla. Entonces ese resultado puede ser utilizado por una consulta principal, se las llama subconsultas o consultas anidadas.

```
SELECT ... FROM ...
WHERE campo1 = (SELECT campo1 FROM ...); -- Debe devolver solo 1 fila o da error

SELECT expr1, ..., (SELECT campo1 FROM ...), ...
FROM ...; -- Debe devolver solo 1 fila o da error

SELECT ...
FROM ...
, (SELECT ... FROM ...) nombre_tabla
-- Puede devolver varias filas y columnas
, ...
```

Se les define un alias para poder luego referenciar en joins, condiciones, o columnas a devolver, etc...

## Operadores de conjunto II

Con operadores de conjunto se puede ver si una condición se cumple contra todos o contra al menos (alguno) de los elementos del conjunto

```
expresion operador [ SOME | ANY | ALL ] (SELECT ...)
WHERE padron = ANY (SELECT padron FROM notas
                     WHERE notas.nota = 10)
```

## Operador IN

Se puede usar para comparar contra un conjunto fijo

```
categoria IN ('A', 'C') -- Similar a un OR
```

O contra el resultado de una subconsulta

```
expr1 IN (SELECT expr2 FROM ...) -- = ANY
expr1 NOT IN (SELECT expr2 FROM ...) -- <> ALL
```

## Operador IN sobre varias expresiones

Es un feature opcional pero que suele estar implementado

```
(expr1, expr2) IN (SELECT expr3, expr4 FROM ...)
```

Revisa que el par de valores este devuelto en al menos una fila de la subconsulta

## Consultas correlacionadas

Una consulta interna puede hacer referencia a tablas de la consulta externa, no al revés.

```
FROM tabla1 t1 WHERE expr1 IN  
(SELECT expr2 FROM tabla2 t2  
WHERE t2.columna2 = t1.columna1)
```

Cuando ocurre decimos que las consultas están correlacionadas. El costo de este tipo de subconsultas suele ser mucho mas alto

## Comparar valores vs comparar conjuntos

Siempre hay que tener cuidado cuando comparamos valores vs conjuntos de valores. Devolver el padrón de los alumnos que no se llamen Lucas

```
SELECT padron FROM alumnos  
WHERE nombre <> 'Lucas';
```

Devolver el padrón de los alumnos que no tienen nota en materias del departamento de código 75

```
-- ESTO ESTA MAL  
SELECT DISTINCT padron FROM NOTAS  
WHERE codigo <> 75;
```

Esto devuelve los alumnos que tienen al menos una nota de un departamento que no es el 75. Si tiene una nota del 75 y otra del 71, lo devuelve por esta ultima, cuando no debería.

```
-- Froma correcta  
SELECT padron FROM alumnos  
WHERE padron NOT IN ( -- o <> ALL  
    SELECT padron FROM notas  
    WHERE codigo = 75  
);
```

## Operador EXISTS

El operador EXISTS devuelve verdadero cuando la subconsulta devuelve al menos una fila. Se suele usar en consultas correlacionadas

```
[NOT] EXISTS (SELECT ...)  
  
-- Por ejemplo  
FROM alumnos a WHERE NOT EXISTS (  
    SELECT 1 -- No importa lo que devuelva, puede ser *  
    FROM notas n  
    WHERE n.padron = a.padron
```

```
    AND n.nota = 10  
);
```

## Agregación

Ciertas consultas, como el promedio de notas de un alumno, precisan los valores de un conjunto de filas. Para resolverlas se utilizan funciones de agregación, que operan sobre un grupo de filas y para el grupo devuelven un único valor

- MAX(expr) y MIN(expr): Devuelven el valor máximo o mínimo de la expresión entre las filas del grupo
- SUM(expr) y AVG(expr): Devuelven la suma o el promedio del valor de la expresión entre las filas del grupo
- COUNT([DISTINCT] expr | \*): Con \* devuelve la cantidad de filas del grupo. Si no, devuelve la cantidad de filas en las que la expresión no es nula. Si se utiliza la opción DISTINCT no cuenta mas de una vez valores repetidos

Todas las filas que hubieran sido devueltas forman un grupo, y se devuelve un único valor para el grupo. Usar una función de agregación entonces modifica la cantidad de filas devueltas y no pueden utilizarse como condición en un WHERE ya que esta se evalúa fila a fila.

## Agrupamiento

Se puede querer usar funciones de agregación pero devolviendo mas de un valor en la consulta

Torneo	Año	Ganador	Premio
Australian Open	2023	Novak Djokovic	2,975,000
French Open	2023	Novak Djokovic	2,300,000
Wimbledon	2023	Carlos Alcaraz	2,350,000
US Open	2023	Novak Djokovic	3,600,000
Australian Open	2024	Jack Sinner	3,150,000
French Open	2024	Carlos Alcaraz	2,400,000
Wimbledon	2024	Carlos Alcaraz	2,700,000

Se debe definir como se van a agrupar las filas y devolver resultados de agregación por cada grupo

- Si quiero datos por tenista, agrupo por "ganador" y se formaran tantos grupos con tenistas ganadores haya
- Si quiero datos por torneo, agrupo por torneo

Las filas se agrupan según el valor de una o mas expresiones utilizando GROUP BY, luego del WHERE y antes del ORDER BY

```
GROUP BY expresion1 [, expresion2, ..., expresionN ]
```

Filas con mismo valor en todas las expresiones estarán en un mismo grupo

Algunos motores permiten acceder a todas las columnas de una tabla si se agrupo por la clave primaria de la tabla. Todas las filas del grupo tendrán el mismo valor en esas columnas

```
SELECT a.padron, a.nombre, ...
FROM alumnos a, ...
GROUP BY a.padron;
```

*Si el motor no lo permite, agrupar por la clave y también por las otras columnas que se quieren devolver.*

*Para contar cuantos ganadores distintos hay:*

```
SELECT Torneo, COUNT(DISTINCT Ganador)
FROM Torneos
GROUP BY Torneo;
```

*Agrupando por ganador, se forman tres grupos*

```
SELECT Ganador, COUNT(*) AS "Ganados", SUM(Premio) AS "Acumulado"
FROM Torneos
GROUP BY Ganador;
```

*Al estar devolviendo una fila por grupo, no se puede devolver directamente columnas por las que no se haya agrupado*

```
-- ESTO ESTA MAL
SELECT Ganador, Año
FROM Torneos
GROUP BY Ganador
```

*las filas de un mismo ganador podrían tener distintos años, como pasa con Carlos Alcaraz. IMPORTANTE: Si se pueden acceder mediante una función de agregación, ya que el valor devuelto dependerá de todos los valores de las columnas entre las filas del mismo grupo*

*Algunos motores permiten acceder a todas las columnas de una tabla si se agrupo por la clave primaria de la tabla*

```
SELECT a.padron, a.nombre, ...
FROM alumnos a, ...
GROUP BY a.padron, ...
```

*Si el motor no lo permite, agrupar por la clave y también por las columnas que se quieren devolver*

## Filtrado de Grupos

Se puede querer no devolver todos los grupos, sino solamente algunos que cumplan cierta condición. Se utiliza la cláusula HAVING luego del GROUP BY, que de modo similar al WHERE tiene una condición a cumplir por el grupo. En la condición se tiene acceso a los datos del grupo, pero no de las filas y se devuelven grupos cuya condición se evalúe como verdadera

```
GROUP BY expresion1 [, expresion2, ..., expresionN ]
HAVING condicion
```

*Si busco los que ganaron mas de un torneo, Sinner no es devuelto*

```
SELECT Ganador, COUNT(*) AS "Ganados", SUM(Premio) AS "Acumulado"
FROM Torneos
```

```
GROUP BY Ganador  
HAVING COUNT(*) > 1;
```

## ▼ V - SQL II

### ABM

#### Inserción

*El comando INSERT se utiliza para insertar filas en una tabla, pudiendo insertarse varias filas a la vez.*

```
INSERT INTO nombre_tabla  
[ (nombre_col1, ..., nombre_colN) ]  
VALUES (valor1, valor2, ..., valorN)  
[, ( valor1, valor2, ..., valorN) ... ]  
;
```

*Se revisan todas las reglas de integridad y si alguna inserción falla, se cancela el total de las inserciones.*

*También puede insertarse el resultado de una consulta.*

```
INSERT INTO nombre_tabla  
[ (nombre_col1, ..., nombre_colN) ]  
SELECT ...;
```

*Con las opción RETURNING (no standard) se permite devolver valores de las filas insertadas que es útil para ver que ids secuenciales se generaron.*

#### Modificación

*Para actualizar fila/s de la tabla se utiliza UPDATE.*

```
UPDATE nombre_tabla SET  
columna1 = expresion1  
[, columna2 = expresion2 ... ]  
[ WHERE condicion ];
```

*sólomente se actualizan las filas que cumplen la condición. En caso de no indicarse condición se actualiza toda la tabla.*

#### Borrado

*El borrado se hace con el comando DELETE.*

```
DELETE FROM nombre_tabla  
[ WHERE condicion ];
```

*Siempre tener cuidado con poner la condición!!!*

#### Transacciones

Cuando queremos que varias operaciones se ejecuten únicamente en su totalidad, usamos transacciones, indicando con comandos el inicio y fin de la transacción.

```
BEGIN TRANSACTION;  
comando1;  
[ comando2; ... comandoN ]  
COMMIT;
```

Se pueden usar niveles de aislamiento. El comando ROLLBACK permite volver atrás los cambios. Se pueden definir SAVEPOINTES hacia los que volver.

## Vistas

Son tablas virtuales. Se las puede acceder como si fueran una tabla mas pero no existe físicamente con una tabla, sus datos surgen como una consulta sobre una o mas tablas.

```
CREATE VIEW nombre_vista  
[ (columna1, columna2, ..., columnaN) ]  
AS consulta_SQL;
```

La consulta no debería tener ORDER BY, aunque algunos SGBD lo permiten como orden por defecto.

Sirven para:

- **Seguridad:** No mostrar ciertas filas/columnas.
- **Reducción de complejidad:** Representar consultas complejas.
- **Cambios de representación de datos:** Usar vistas con versión vieja.

## Actualización de datos en vistas

Algunos gestores permite actualizaciones en vistas, siguiendo algunas restricciones:

- La vista tiene una única tabla en el FROM.
- La vista no usa operaciones de conjunto.
- La vista no usa agrupamiento.
- La vista no limita la cantidad de resultados.
- Solo se pueden actualizar valores de columnas accedidas directamente.
- Las restricciones de la tabla se deben cumplir, o la actualización fallara.

## Problemas con filas que quedan fuera de la vista

Si la consulta de la vista tiene un WHERE y se crean o modifican filas, pueden quedar sin cumplir la condición y quedar fuera de la vista. PostgreSQL permite impedir este tipo de operaciones si se define la vista con la opción CHECK (no standard).

```
CREATE VIEW nombre_vista  
[ (columna1, columna2, ..., columnaN) ]  
AS consulta_SQL  
WITH CHECK OPTION;
```

## WITH y WITH RECURSIVE

La cláusula WITH (opcional en el estándar SQL) permite definir una consulta auxiliar temporal a utilizar durante una consulta. Se puede pensar como una tabla temporal, que existe únicamente en la ejecución de la consulta. Permite evitar repetir una estructura varias veces y asociar a un nombre para tener una consulta más clara. Usar con cuidado porque exceso de WITHs puede complejizar una consulta sencilla.

```
WITH alias1 [ (col1, ..., colN] AS (subconsulta)
      [, alias2 [ (col1b, ..., colNb] AS (subconsulta2) ]
consulta_principal
```

```
-- Ejemplo
WITH aprobados AS (SELECT * FROM notas WHERE nota >= 4)
SELECT padron
FROM aprobados
GROUP BY padron
HAVING AVG(nota) >= ALL (
    SELECT AVG(nota)
    FROM aprobados
    GROUP BY padron
);
```

## WITH RECURSIVO

La opción RECURSIVE permite que una subconsulta se refiera a si misma. Esto es útil para resolver consultas iterativas.

Teniendo la tabla "vuelos" que indican ciudades origen y destino, ¿a qué ciudades puedo llegar desde una en particular, haciendo todas las escalas que se necesiten?

codigo	desde	hasta
444	París	Madrid
510	París	Roma
610	Brasilia	París
690	Brasilia	Montevideo
880	Roma	Amsterdam
900	Amsterdam	Oslo

```
WITH RECURSIVE alias [ (col1, ..., colN) ]
AS (
    SELECT ... FROM ...
    UNION
    SELECT ... FROM ...
```

```
)  
SELECT ... FROM ...
```

Se comienza con la subconsulta fija y se ejecuta iterativamente la unión con la subconsulta recursiva hasta que no se agreguen nuevas filas en una iteración.

```
-- Consigue todas las ciudades alcanzadas desde Paris  
WITH RECURSIVE trayectos (ciudad) AS (  
    SELECT hasta FROM vuelos  
    WHERE desde = 'Paris'  
    UNION  
    SELECT hasta FROM vuelos v  
    INNER JOIN trayectos t ON v.desde = t.ciudad  
)  
SELECT * FROM trayectos  
  
-- Otra versión  
WITH RECURSIVE alcanzables AS (  
    SELECT hasta FROM vuelos  
    WHERE desde = 'Paris'  
    UNION  
    SELECT v.hasta FROM alcanzables a, vuelos v  
    WHERE a.hasta = v.desde  
)  
SELECT * FROM alcanzables;
```

## Control de acceso a los datos

En base de datos multiusuarios, muchas veces es necesario restringir el acceso a los datos. Distintos usuarios deben tener acceso a distintos subconjuntos de los datos:

- No todas las tablas
- No todas las columnas
- No todas las filas

También se debe poder restringir operaciones a realizar. No todos deberían poder modificar/crear/insertar datos en la base.

En general se puede restringir desde que redes se aceptan conexiones, en algún archivo de configuración . Una vez aceptada la conexión, los accesos se gestionan por usuario y por roles a los que pertenecen los usuarios.

Crear un usuario es crear un rol con login.

```
CREATE USER nombre_usuario WITH PASSWORD 'pass';  
  
CREATE ROLE nombre_usuario WITH LOGIN PASSWORD 'pass';
```

Se pueden quitar usuarios/roles con el comando *DROP*.

```
DROP ROLE nombre_usuario;
```

Se recomienda gestionar los permisos en roles y asignarle roles a usuarios, para agrupar perfiles comunes.

```
CREATE ROLE nombre_rol;
```

```
GRANT nombre_rol TO nombre_usuario;
```

Se pueden desasignar roles a usuarios.

```
REVOKE nombre_rol FROM nombre_usuario;
```

El comando GRANT permite dar privilegios a roles.

```
-- privilegio_i in { SELECT, DELETE, INSERT, TRIGGER, UPDATE, CREATE }
GRANT privilegio1 [, privilegio2, ..., privilegioN ]
    ON [nombre_tabla | DATABASE nombre_base_de_datos ]
    TO nombre_rol
    [ WITH GRANT OPTION ];
```

La opción GRANT OPTION permite que el rol también pueda dar los privilegios a otros roles utilizando GRANT.

## Quita de privilegios

Para quitar privilegios se utiliza el comando REVOKE.

```
REVOKE [ GRANT OPTION FOR ]
    privilegio1 [, privilegio2, ..., privilegioN ]
    ON [ nombre_tabla | DATABASE nombre_base_de_datos ]
    FROM nombre_rol;
```

## Uso de vistas para restringir acceso

Lo anterior da privilegios a todos los datos de las tablas, para restringir a ciertas columnas o ciertas filas, se pueden utilizar vistas de SQL y dar permisos sobre ellas. No incluir todas las columnas o no incluir todas las filas.

## ▼ VI - Diseño Relacional I

### Formas Normales

Criterios de un buen diseño relacional

- Preservación de información
- Redundancia mínima

Cuando se parte de un correcto diseño conceptual y se hace un correcto pasaje al modelo lógico, se obtiene un esquema sin redundancia y se preserva toda la información del mundo real que se quería modelar.

### Dependencia Funcional

Dada una relación  $R(A)$ , una dependencia funcional  $X \rightarrow Y$ , con  $X, Y$  incluidos en  $A$  es una restricción sobre las posibles tuplas de  $R$  que implica que dos tuplas con igual valor del conjunto de atributos  $X$  deben también tener igual valor del conjunto de atributos  $Y$ .

La dependencia funcional  $X \rightarrow Y$  implica que hay una relación funcional entre los valores de  $X$  y los de  $Y$  dentro de la base de datos.

## Formas Normales

Las formas normales son una serie de estructuras con las que un esquema de base de datos puede cumplir o no. Las formas normales clásicas son:

- Primera forma normal (1FN)
- Segunda forma normal (2FN)
- Tercera forma normal (3FN)
- Forma normal Boyce-Codd (FNBC)
- Cuarta forma normal (4FN)
- Quinta forma normal (5FN)

Cada forma normal es mas fuerte que las anteriores en el orden en que las hemos introducido.

La normalización es el proceso a través del cual se convierte un esquema de base de datos en uno equivalente (i.e., que preserva toda la información) y que cumple con una determinada forma normal.

El objetivo es:

- Preservar la información
- Eliminar la redundancia
- Evitar las anomalías de ABM

### Primer forma normal (1FN)

Decimos que un esquema de base de datos relacional esta en primera forma normal (1FN) cuando los dominios de todos sus atributos solo permiten valores atómicos (es decir, indivisibles) y monovaluados. Actualmente, se considera que en el modelo relacional todos los atributos deben ser monovaluados y atómicos. Con este criterio, todo esquema relacional esta ya en 1FN.

#### ■ Situación:

nombre_profesor	mail
Juan Gómez	{jgomez@udbc.com.ar, jgomez94@mibase.com}
Roberta Casas	{rcasas@udbc.com.ar, rcasas@ggmail.com}
Irene Adler	{iadler@udbc.com.ar}

#### ■ Solución 1: Colocar un mail por tupla y repetir el nombre del profesor.

nombre_profesor	mail
Juan Gómez	jgomez@udbc.com.ar
Juan Gómez	jgomez94@mibase.com
Roberta Casas	rcasas@udbc.com.ar
Roberta Casas	rcasas@ggmail.com
Irene Adler	iadler@udbc.com.ar

#### ■ Solución 2: Suponer un máximo posible $M$ de mails y tener $M$ atributos distintos reservados a tal fin. Para profesores que tienen menos de $M$ mails, quedarán valores nulos.

nombre_profesor	mail1	mail2
Juan Gómez	jgomez@udbc.com.ar	jgomez94@mibase.com
Roberta Casas	rcasas@udbc.com.ar	rcasas@ggmail.com
Irene Adler	iadler@udbc.com.ar	NULL

))

### Segunda forma normal (2FN)

D P A

- Consideremos ahora el siguiente ejemplo, ya en 1FN:

nombre_dpto	nombre_profesor	asignatura
Física	Juan Gómez	Física II
Física	Roberta Casas	Física II
Física	Juan Gómez	Física III
Matemática	Roberta Casas	Topología
Matemática	Irene Adler	Álgebra I

- Identifiquemos las dependencias funcionales semánticas:
  - $\text{asignatura} \rightarrow \text{nombre\_dpto}$
- Existen otras dependencias funcionales que pueden deducirse de la anterior:
  - $\{\text{nombre\_profesor}, \text{asignatura}\} \rightarrow \text{nombre\_dpto}$
- Y otras que son triviales:
  - Ejemplo:  $\{\text{nombre\_profesor}, \text{asignatura}\} \rightarrow \text{asignatura}$

Una dependencia funcional  $X \rightarrow Y$  es parcial cuando existe un subconjunto propio  $A$  incluido en  $X$ ,  $A \neq X$  para el cual  $A \rightarrow Y$ .

Una dependencia funcional  $X \rightarrow Y$  es completa si no es parcial.

En el ejemplo,  $\text{nombre\_dpto}$  tiene dependencia funcional completa de la clave primaria  $\{\text{nombre\_profesor}, \text{asignatura}\}$ .

Atributo primo en una relación: Es aquel que es parte de alguna clave candidata de la relación.

Decimos que una relación está en segunda forma normal (2FN) cuando todos sus atributos no primos tienen dependencia funcional completa de las claves candidatas.

nombre_dpto	nombre_profesor	asignatura
Física	Juan Gómez	Física II
Física	Roberta Casas	Física II
Física	Juan Gómez	Física III
Matemática	Roberta Casas	Topología
Matemática	Irene Adler	Álgebra I

- ¿Cómo resolvemos la situación en el ejemplo?
  - DocenteAsignatura(nombre\_profesor, asignatura)
  - AsignaturaDepartamento(asignatura, nombre\_dpto)

## Inferencia de dependencias funcionales

Axiomas:

- Reflexividad:  $Y$  incluido en  $X \Rightarrow X \rightarrow Y$
- Aumento: Para todo  $W$ :  $X \rightarrow Y \Rightarrow XW \rightarrow YW$
- Transitividad:  $X \rightarrow Y$  and  $Y \rightarrow Z \Rightarrow X \rightarrow Z$

*Estos axiomas pueden ser probados a partir de la definición de su dependencia funcional (i.e., no son axiomas en stricto sensu).*

*Los tres axiomas en conjunto son completos: Toda dependencia funcional que se puede inferir de F se puede inferir a través de los axiomas de Armstrong.*

*La notación  $F \vdash X \rightarrow Y$  indica que la dependencia funcional  $X \rightarrow Y$  puede inferirse a partir del conjunto de dependencias funcionales F.*

**Ejercicio:** Muestre que dado el conjunto de dependencias funcionales  $F = \{ A \rightarrow C, BC \rightarrow E, D \rightarrow B \}$  es posible inferir que  $AD \rightarrow E$ .

$DC \rightarrow BC$  (aumento, agrego C en ambos lados de  $D \rightarrow B$ )  
 $AD \rightarrow DC$  (aumento, agrego D en ambos lados de  $A \rightarrow C$ )  
 $AD \rightarrow BC$  (por transitividad)  
 $AD \rightarrow E$

*Axiomas de Armstrong:*

- *Regla de unión:  $X \rightarrow Y$  and  $X \rightarrow Z \Rightarrow X \rightarrow YZ$*
- *Regla de pseudotransitividad: Para todo W:  $X \rightarrow Y$  and  $YW \rightarrow Z \Rightarrow XW \rightarrow Z$*
- *Regla de descomposición:  $X \rightarrow YZ \Rightarrow X \rightarrow Y$  and  $X \rightarrow Z$*

## Clausuras de conjuntos de Dependencias Funcionales y de Atributos

*Dado un conjunto de dependencias funcionales F, la clausura de F ( $F^+$ ) es el conjunto de todas las dependencias funcionales que pueden inferirse de F.*

*Dado un conjunto de atributos X y un conjunto de dependencias F, la clausura de X con respecto a F ( $XF^+$ ) es el conjunto de todos los atributos  $A_i$  tales que la dependencia funcional  $X \rightarrow A_i$  se infiere del conjunto de dependencias F.*

## Descomposición

*Partimos del concepto de relación universal: Una relación  $R(A_1, A_2, \dots, A_n)$  que engloba todos los atributos del mundo real que nuestro modelo lógico representa. Dada una relación universal y un conjunto de dependencias funcionales F definidas sobre ella, decimos que un conjunto de relaciones es una descomposición de R cuando todos los atributos de la relación R se conservan.*

*Analizaremos dos propiedades de las descomposiciones:*

- *La preservación de información*
- *La preservación de dependencias funcionales*

*Si una descomposición cumple que para toda instancia posible de R, la junta de las proyecciones sobre los  $R_i$  permite recuperar la misma instancia de relación, entonces decimos que la descomposición preserva la información.*

Diremos que la descomposición preserva las dependencias funcionales cuando toda dependencia funcional  $X \rightarrow Y$  en  $R$  puede inferirse a partir de dependencias funcionales definidas en los  $R_i$ .

## Tercera Forma Normal (3FN)

- Veamos el siguiente ejemplo:

VENTAS						
nro_factura	cliente	nro_item	cod_producto	nombre_producto	cantidad	precio_unit
0003-45821	Lionel Pessari	1	249	Suprabond 500mg	2	87.00
0003-45821	Lionel Pessari	2	230	Tersuave azul 4l	1	270.00
0003-45821	Lionel Pessari	3	115	Brocha 5cm	2	90.00
0003-45822	Claudia Serrano	1	258	Alba p/Exteriores 3l	2	225.00
0003-45822	Claudia Serrano	2	116	Brocha 10cm	2	130.00
0003-45823	Claudia Serrano	1	330	Cetol 2l	1	315.00

- Identificamos las dependencias funcionales no triviales a partir de la semántica:
  - $nro\_factura \rightarrow cliente$
  - $\{nro\_factura, nro\_item\} \rightarrow \{nombre\_producto, cod\_producto, cantidad, precio\_unit\}$
  - $cod\_producto \rightarrow nombre\_producto$
- Identificamos la clave primaria:
  - $\{nro\_factura, nro\_item\}$
- No hay otras claves candidatas.

CLIENTE FACTURA	
nro_factura	cliente
0003-45821	Lionel Pessari
0003-45822	Claudia Serrano

DETALLE FACTURA					
nro_factura	nro_item	cod_producto	nombre_producto	cantidad	precio_unit
0003-45821	1	249	Suprabond 500mg	2	87.00
0003-45821	2	230	Tersuave azul 4l	1	270.00
0003-45821	3	115	Brocha 5cm	2	90.00
0003-45822	1	258	Alba p/Exteriores 3l	2	225.00
0003-45822	2	116	Brocha 10cm	2	130.00
0003-45823	1	330	Cetol 2l	1	315.00

- Observemos que todas las dependencias funcionales que había se mantienen.
- Sin embargo, una de las dependencias muestra que un atributo **no primo** puede deducirse a partir de otro atributo no primo.
  - $cod\_producto \rightarrow nombre\_producto$
- Entonces, decimos que  $nombre\_producto$  tiene “dependencia transitiva” en la clave primaria, lo que no es deseable.

Para toda dependencia funcional no trivial  $X \rightarrow Y$ , o bien  $X$  es superclave, o bien  $Y - X$  contiene solo atributos primos.

- Volvamos al ejemplo de los tenistas:

nombre_torneo	año	PARTIDOS			set	punt1	punt2	NP	NP
		tenista1	tenista2	ronda					
Roland Garros	2016	A. Murray	S. Wawrinka	2-final	1	6	4		
Roland Garros	2016	A. Murray	S. Wawrinka	2-final	2	6	2		
Roland Garros	2016	A. Murray	S. Wawrinka	2-final	3	4	6		
Roland Garros	2016	A. Murray	S. Wawrinka	2-final	4	6	2		
Masters de Madrid	2015	R. Federer	R. Nadal	4-final	1	3	6		
Masters de Madrid	2015	R. Federer	R. Nadal	4-final	2	1	6		
Roland Garros	2016	N. Djokovic	A. Murray	Final	1	6	3		
Roland Garros	2016	N. Djokovic	A. Murray	Final	2	1	6		
Roland Garros	2016	N. Djokovic	A. Murray	Final	3	6	2		
Roland Garros	2016	N. Djokovic	A. Murray	Final	4	6	4		

- ¿Hay dependencias transitivas de atributos no primos? No.

- $\{ \text{nombre\_torneo}, \text{año}, \text{tenista1}, \text{tenista2}, \text{set} \} \rightarrow \text{punt1}$
- A su vez:  $\{ \text{nombre\_torneo}, \text{año}, \text{tenista1}, \text{tenista2} \} \rightarrow \text{ronda}$
- Y luego:  $\{ \text{nombre\_torneo}, \text{año}, \text{tenista1}, \text{ronda}, \text{set} \} \rightarrow \text{punt1}$
- Pero *ronda* es parte de una clave candidata.

- Por lo tanto, está en tercera forma normal.

## Forma Normal Boyce-Codd (FNBC)

- Hay un tipo de redundancia que aún no eliminamos...

A	CURSADA M	P
Dante Micelli	Zoología	Edmundo Ribeiro
Dante Micelli	Botánica	José Cestoni
Dante Micelli	Anatomía General I	Pedro González
Alberto Deheza	Botánica	José Cestoni
Alberto Deheza	Zoología	Viviana Díaz
Carla Hernández	Zoología	Edmundo Ribeiro
Carla Hernández	Anatomía General I	Pedro González
Carla Hernández	Botánica	José Cestoni
Leticia Humboldt	Botánica	Héctor Larraza
Leticia Humboldt	Zoología	Viviana Díaz

Hipótesis: Cada materia es dictada por muchos profesores, pero un estudiante sólo cursa con uno de ellos. La universidad tiene la restricción de que un profesor sólo puede dictar una materia.

- Identificamos las dependencias funcionales no triviales a partir de la semántica:

$$f = \cancel{\exists} \{AM \rightarrow P\} \quad \leftarrow \langle AM \rangle \langle AP \rangle$$

INSCRIPCIONES	
alumno	profesor
Dante Micelli	Edmundo Ribeiro
Dante Micelli	José Cestoni
Dante Micelli	Pedro González
Alberto Deheza	José Cestoni
Alberto Deheza	Viviana Díaz
Carla Hernández	Edmundo Ribeiro
Carla Hernández	Pedro González
Carla Hernández	José Cestoni
Leticia Humboldt	Héctor Larraza
Leticia Humboldt	Viviana Díaz

CURSOS	
materia	profesor
Zoología	Edmundo Ribeiro
Botánica	José Cestoni
Anatomía General I	Pedro González
Zoología	Viviana Díaz
Botánica	Héctor Larraza

## ▼ VII - Diseño Relacional II

**Atributo Primo:** Pertenece a una clave candidata

**2FN:** No hay dependencia parcial para atributos no primos sobre las claves candidatas.

**3FN:** Para toda dependencia  $X \rightarrow Y$  no trivial, o  $X$  es superclave o  $\{Y - X\}$  son atributos primos.

**FNBC:** Para toda dependencia  $X \rightarrow Y$  no trivial,  $X$  es superclave de  $R$ .

## Equivalencia de conjuntos de dependencias

$R(A, B, C)$

$F1 = \{ A \rightarrow B, B \rightarrow C \}$

$F2 = \{ A \rightarrow B, B \rightarrow C, A \rightarrow C \}$

$F1$  y  $F2$  son equivalentes

## Algoritmo de Cubrimiento Minimal

Dados dos conjuntos de dependencias funcionales  $F$  y  $G$ , decimos que el conjunto  $F$  cubre a  $G$  cuando toda dependencia  $X \rightarrow Y$  in  $G$  puede ser inferida a partir de  $F$ .

Dos conjuntos de dependencias funcionales  $F$  y  $G$  son equivalentes cuando cada uno de ellos es cubierto por el otro. En otras palabras,  $F$  y  $G$  son equivalentes cuando sus clausuras coinciden:  $F^+ = G^+$ .

### Ejemplo:

Muestre que los conjuntos de dependencias funcionales  $F1 = \{ A \rightarrow B, B \rightarrow C, C \rightarrow A \}$  y  $F2 = \{ A \rightarrow C, C \rightarrow B, B \rightarrow A \}$  son equivalentes.

Dado un conjunto de dependencias  $F$ , nos interesa encontrar un conjunto equivalente  $G$  que cumpla ciertas propiedades de minimalidad. En particular, nos interesa que:

- No haya atributos innecesarios del lado izquierdo
- No haya dependencias redundantes

A todo conjunto de dependencias funcionales  $G$  que es equivalente a  $F$  y cumple estas dos propiedades lo denominamos cubrimiento minimal de  $F$ .

El algoritmo de cubrimiento minimal tiene 3 grandes pasos:

1. Pasar las dependencias funcionales a forma canónica (descomponer cada dependencia funcional  $X \rightarrow Y$  en dfs  $X \rightarrow A_i$  con  $A_i$  in  $Y$ ).
2. Eliminar los atributos innecesarios del lado izquierdo de cada dependencia funcional  $X \rightarrow A_i$
3. Eliminar las dependencias funcionales redundantes.

**Ejemplo:**

$R(A, B, C, D, E, F, G)$

$F = \{ AB \rightarrow C, C \rightarrow A, BC \rightarrow D, ACD \rightarrow B, D \rightarrow EG, BE \rightarrow C, CG \rightarrow BD, CD \rightarrow AG \}$

**Paso 1:** Que quede un solo atributo del lado derecho

$F1 = \{ AB \rightarrow C, C \rightarrow A, BC \rightarrow D, ADC \rightarrow B, D \rightarrow E, D \rightarrow G, BE \rightarrow C, CG \rightarrow B, CG \rightarrow D, CE \rightarrow A, CE \rightarrow G \}$

**Paso 2:** Quitar atributos que sobren del lado izquierdo

$A^+ = \{ A \}$

$B^+ = \{ B \}$

$C^+ = \{ A, C \}$

$D^+ = \{ D, E, G \}$

$\{ C, D \}^+ = \{ A, B, C, D, E, G \}$

$E^+ = \{ E \}$

$G^+ = \{ G \}$

$F2 = \{ AB \rightarrow C, C \rightarrow A, BC \rightarrow D, \mathbf{CD \rightarrow B}, D \rightarrow E, D \rightarrow G, BE \rightarrow C, CG \rightarrow B, CG \rightarrow D, \mathbf{C \rightarrow A}, CE \rightarrow G \}$

**Paso 2.2:** Sacamos repetidos

$F2.2 = \{ AB \rightarrow C, C \rightarrow A, BC \rightarrow D, CD \rightarrow B, D \rightarrow E, D \rightarrow G, BE \rightarrow C, CG \rightarrow B, CG \rightarrow D, CE \rightarrow G \}$

**Paso 3:** Eliminar dependencias funcionales redundantes

$F3 = \{ AB \rightarrow C, C \rightarrow A, BC \rightarrow D, \mathbf{CD \rightarrow B}, D \rightarrow E, D \rightarrow G, BE \rightarrow C, CG \rightarrow B, CG \rightarrow D, CE \rightarrow G \}$

**IMPORTANTE:** Apenas detectamos redundancia la sacamos de  $F3$

$(\{ A, B \} + (F3 - AB \rightarrow C))^+ = \{ A, B \}, AB \rightarrow C \text{ no es redundante}$

$(\{ C \} + (F3 - C \rightarrow A))^+ = \{ C \}, C \rightarrow A \text{ no es redundante}$

$(\{ B, C \} + (F3 - BC \rightarrow D))^+ = \{ A, B, C \}, BC \rightarrow D \text{ no es redundante}$

$(\{ C, D \} + (F3 - CD \rightarrow B))^+ = \{ A, B, C, D, E, G \}, CD \rightarrow B \text{ es redundante}$

$(\{ D \} + (F3 - D \rightarrow E))^+ = \{ D, G \}, D \rightarrow E \text{ no es redundante}$

$(\{ D \} + (F3 - D \rightarrow G))^+ = \{ D, E \}, D \rightarrow G \text{ no es redundante}$

$(\{ B, E \} + (F3 - BE \rightarrow C))^+ = \{ B, E \}, BE \rightarrow C \text{ no es redundante}$

$(\{C, G\} + (F3 - CG \rightarrow B)) + = \{A, C, D, E, G\}$ ,  $CG \rightarrow B$  no es redundante

$(\{C, G\} + (F3 - CG \rightarrow D)) + = \{A, B, C, D, E, G\}$ ,  $CG \rightarrow D$  es redundante

$(\{C, E\} + (F3 - CE \rightarrow G)) + = \{A, C, E\}$ ,  $CE \rightarrow G$  no es redundante

Por lo que  $F_{min} = \{AB \rightarrow C, C \rightarrow A, BC \rightarrow D, D \rightarrow E, D \rightarrow G, BE \rightarrow C, CG \rightarrow B, CE \rightarrow G\}$

## Algoritmo de búsqueda de claves candidatas

$R(A, B, C, D, E, F, G, H, I, J)$

$F = \{AB \rightarrow C, A \rightarrow DE, B \rightarrow F, F \rightarrow GH, D \rightarrow IJ, B \rightarrow A, H \rightarrow G\}$

$F_{min} = \{AB \rightarrow C, A \rightarrow D, A \rightarrow E, B \rightarrow F, F \rightarrow H, D \rightarrow I, D \rightarrow J, B \rightarrow A, H \rightarrow G\}$

No hace falta que sea minimal pero lo hace mas sencillo

Atributo independiente: No aparece en ninguna DF  $\Rightarrow$  Esta en todas las CCs

Atributo que solo aparece en lados izquierdos  $\Rightarrow$  Esta en todas las CCs

Atributo que solo aparece en lados derechos  $\Rightarrow$  No están en ninguna CC

Atributo que aparece en lados izquierdos y derechos  $\Rightarrow$  ???

A	B	C	D	E	F	G	H
I	I		I		I		I
D		D	D	D	D	D	D

Solo en izquierda: B

Solo en derecha: C, E, G, I, J

Ambos lados: A, D, F, H

$F_{min} = \{AB \rightarrow C, A \rightarrow D, A \rightarrow E, B \rightarrow F, F \rightarrow H, D \rightarrow I, D \rightarrow J, B \rightarrow A, H \rightarrow G\}$

$B^+ = \{A, B, C, D, E, F, G, H, I\} = R \Rightarrow CCs = \{\{B\}\}$

### Segundo ejemplo:

$R(A, B, C, D, E, F, G)$

$F_{min} = \{AB \rightarrow F, D \rightarrow A, E \rightarrow D, D \rightarrow E, CF \rightarrow B, B \rightarrow C\}$ , **D es equivalente a E**

$R_{aux}(A, B, C, E, F, G)$

$F_{minaux} = \{AB \rightarrow F, E \rightarrow A, CF \rightarrow B, B \rightarrow C\}$

A	B	C	E	F	G
I	I	I	I	I	
D	D	D		D	

Atributos independientes: G  $\Rightarrow$  Esta en todas las claves

Solo en izquierda: E  $\Rightarrow$  Va a estar en todas las claves

Solo ambos lados: A, B, C, F

$F_{minaux} = \{ AB \rightarrow F, E \rightarrow A, CF \rightarrow B, B \rightarrow C \}$

$\{E, G\}^+ = \{A, E, G\}, \text{ no es CC}$

$\{A, E, G\}^+ = \{A, E, G\}, \text{ no es CC}$

$\{B, E, G\}^+ = \{A, B, C, E, F, G\} = R_{aux}, \text{ es CC}$

$\{C, E, G\}^+ = \{C, E, G\}, \text{ no es CC}$

$\{E, F, G\}^+ = \{E, F, G\}, \text{ no es CC}$

Seguimos probando los demás

$\{A, C, E, G\}^+ = \{A, C, E, G\}, \text{ no es CC}$

$\{A, E, F, G\}^+ = \{A, E, F, G\}, \text{ no es CC}$

$\{C, E, F, G\}^+ = \{A, B, C, E, F, G\} = R_{aux}, \text{ es CC}$

$R_{aux}$  tiene dos CCs:  $\{B, E, G\}$  y  $\{C, E, F, G\}$

$R$  tiene 4 CCs:  $\{B, E, G\}, \{B, D, G\}, \{C, E, F, G\}, \{C, D, F, G\}$

### Ejemplo 2:

$R(A, B, C, D, E, F, G, H, I, J)$

$F = \{AB \rightarrow C, BD \rightarrow EF, AD \rightarrow GH, A \rightarrow I, H \rightarrow J\}$

**Paso 1:** Encontrar un  $F_{min}$

$F1 = \{AB \rightarrow C, BD \rightarrow E, BD \rightarrow F, AD \rightarrow G, AD \rightarrow H, A \rightarrow I, H \rightarrow J\}$

$A^+ = \{A, I\}$

$B^+ = \{B\}$

$C^+ = \{C\}$

$D^+ = \{D\}$

$E^+ = \{E\}$

$F^+ = \{F\}$

$G^+ = \{G\}$

$H^+ = \{H, J\}$

$I^+ = \{I\}$

$J^+ = \{J\}$

$F3 = \{AB \rightarrow C, BD \rightarrow E, BD \rightarrow F, AD \rightarrow G, AD \rightarrow H, A \rightarrow I, H \rightarrow J\}$

$(\{A, B\} - (F3 - AB \rightarrow C))^+ = \{A, B, I\}, \text{ no es redundante}$

$(\{B, D\} - (F3 - BD \rightarrow E))^+ = \{B, D, F\}, \text{ no es redundante}$

$(\{B, D\} - (F3 - BD \rightarrow F))^+ = \{B, D, E\}, \text{ no es redundante}$

$(\{A, D\} - (F3 - AD \rightarrow G))^+ = \{A, D, H, I, J\}, \text{ no es redundante}$

$(\{A, D\} - (F3 - AD \rightarrow H))^+ = \{A, D, G\}, \text{ no es redundante}$

$\{A\} - (F3 - A \rightarrow I) + = \{A\}$ , no es redundante

$\{H\} - (F3 - H \rightarrow J) + = \{H\}$ , no es redundante

$F_{min} = \{AB \rightarrow C, BD \rightarrow E, BD \rightarrow F, AD \rightarrow G, AD \rightarrow H, A \rightarrow I, H \rightarrow J\}$

A	B	C	D	E	F	G	H
I	I		I				I
		D		D	D	D	D

Atributos independientes: {}

Atributos solo en izquierda: {A, B, D}, deben ser parte de las CC

Atributos solo en derecha: {C, E, F, G, I, J}, no forman parte de ninguna CC

$\{A, B, D\} + = \{A, B, C, D, E, F, G, H, I, J\}$ , es CC

Como {A, B, D} es CC y es el conjunto que debe estar si o si en todas las CCs, es la única CC.

CCs de R = {{A, B, D}}

## Algoritmo de descomposición a 3FN

$R(A, B, C, D, E, G)$  y  $F_{min} = \{AB \rightarrow C, C \rightarrow B, E \rightarrow D, D \rightarrow G, G \rightarrow E, C \rightarrow G\}$

**Paso 1:** Calcular CCs

A	B	C	D	E	G
I	I	I	I	I	I
	D	D	D	D	D

Independientes: {}

Equivalentes: {E, D, G}, no es necesario quitarlos pero podemos quedarnos con uno de ellos

Solo izquierda: {A}, están en todas las CC

Solo derecha: {}, no están en ninguna CC

$R_{aux} = \{A, B, C, E\}$

$F_{minaux} = \{AB \rightarrow C, C \rightarrow B, C \rightarrow E\}$

A	B	C	E
I	I	I	
	D	D	D

Solo izquierda: {A}, están en todas las CC

Solo derecha: {E}, no están en ninguna CC

$A^+ = \{A\}$ , no es CC

$\{A, B\}^+ = \{A, B, C, E\}$ , es CC

$\{A, C\}^+ = \{A, B, C, E\}$ , es CC

$\{A, E\}^+ = \{A, E\}$ , no es CC

CCs de  $R_{aux}$ :  $\{\{A, B\}, \{A, C\}\}$

CCs de  $R$ :  $\{\{A, B\}, \{A, C\}\}$  pues  $E$  no aparece en ninguna CC de  $R_{aux}$

**Paso 2:** Pasar a 3FN, para cada DF obtener un  $R_i$  que tenga a todos sus atributos

$F_{min} = \{AB \rightarrow C, C \rightarrow B, E \rightarrow D, D \rightarrow G, G \rightarrow E, C \rightarrow G\}$

$R1(A, B, C)$

$R2(B, C)$

$R3(D, E)$

$R4(D, G)$

$R5(E, G)$

$R6(C, G)$

**Paso 3:** Si no existe una CC en alguna de estas  $R_i$ , debo agregar otro que tenga alguna de ellas. Por ejemplo en este caso  $R1$  contiene a  $\{A, B\}$  y a  $\{A, C\}$ , ambas son CC de  $R$ , si no existiera  $R1$ , ninguna de las otras  $R_i$  contiene a  $\{A, B\}$  o  $\{A, C\}$  por lo que habría que crear un  $R_7$  tal que  $R7(A, B)$  o  $R7(A, C)$ . **No debe contener ambas, solo una.**

**Paso 3:** Quitar todas las relaciones que estén contenidas en otra.

$R1(A, B, C)$

~~$R2(B, C)$~~ , está contenida en  $R1$

$R3(D, E)$

$R4(D, G)$

$R5(E, G)$

$R6(C, G)$

**Paso 4:** Determinar las CCs de cada una de los  $R_i$

$F_{min} = \{AB \rightarrow C, C \rightarrow B, E \rightarrow D, D \rightarrow G, G \rightarrow E, C \rightarrow G\}$

$R1(A, B, C), F = \{AB \rightarrow C, C \rightarrow B\}, CCs = \{\{A, B\}, \{A, C\}\}$

$R3(D, E), F = \{E \rightarrow D, D \rightarrow E\}, CCs = \{\{E\}, \{D\}\}$

$R4(D, G), F = \{D \rightarrow G, G \rightarrow D\}, CCs = \{\{D\}, \{G\}\}$

$R5(E, G), F = \{G \rightarrow E, E \rightarrow G\}, CCs = \{\{E\}, \{G\}\}$

$R6(C, G), F = \{C \rightarrow G\}, CCs = \{\{C\}\}$

**Paso 5:** Combinar las relaciones que comparten CCs

$R1(A, B, C), F = \{AB \rightarrow C, C \rightarrow B\}, CCs = \{\{A, B\}, \{A, C\}\}$

$R345(D, E, G), F = \{D \rightarrow E, E \rightarrow G, G \rightarrow D\}, CCs = \{\{D\}, \{E\}, \{G\}\}$

$R6(C, G), F = \{C \rightarrow G\}, CCs = \{\{C\}\}$

Está en 3FN, no está en FNBC

## Algoritmo de descomposición a FNBC

$R(A, B, C, D, E, F, G), F = \{ AB \rightarrow C, C \rightarrow B, E \rightarrow D, D \rightarrow G, G \rightarrow E, C \rightarrow G \}$

Descompongo por  $C \rightarrow B$

C+	R - C+ + C
{ B, C, D, E, G }, CCs = { { C } }, no está en FNBC	{ A, C }, CCs = { { A, C } }, FNBC

Descompongo por  $D \rightarrow G$  desde { B, C, D, E, G }

D+	{ B, C, D, E, G } - D+ + D
{ D, E, G }, CCs = { { D }, { E }, { G } }, FNBC	{ B, C, D }, CCs = { { C } }, FNBC

R1(A, C) con CCs { { A, C } } sin dfs en FNBC

R2(D, E, G) con  $E \rightarrow D, D \rightarrow G, G \rightarrow E$  con CCs { { E }, { D }, { G } } en FNBC

R3(B, C, D) con  $C \rightarrow B, C \rightarrow D$  con CCs { { C } } en FNBC

**AB+ = { A, B }, no llega a C, por lo que  $AB \rightarrow C$  no existe**

C+ = { B, C, D, G }, llega a G, por lo que  $C \rightarrow G$  existe

{  $AB \rightarrow C, C \rightarrow B, E \rightarrow D, D \rightarrow G, G \rightarrow E, C \rightarrow G$  }

## ▼ VIII - Concurrencia y Transacciones

### Sistemas monoprocesador

Permiten hacer multitasking (multitarea). Varios hilos o procesos pueden estar corriendo concurrentemente.

### Sistemas multiprocesador y sistemas distribuidos

Disponen de varias unidades de procesamiento que funcionan en forma simultanea. Suelen replicar la base de datos, disponiendo de varias copias de algunas tablas (o fragmentos de tabla) en distintas unidades de procesamiento.

### Transacción

En este contexto utilizaremos el concepto de transacción como "unidad lógica de trabajo" o, más en detalle, "secuencia ordenada de instrucciones atómicas".

Una misma transacción puede realizar varias operaciones de consulta/ABM durante su ejecución. Antes de existir el multitasking, las transacciones se serializaban. Hasta tanto no se terminara una, no se iniciaba la siguiente. Serializar es en general una mala idea. Nos gustaría poder ejecutarlas en forma simultanea, aunque garantizando ciertas propiedades básicas.

### Concurrencia

*La concurrencia es la posibilidad de ejecutar múltiples transacciones (tareas, en la jerga de sistemas operativos) en forma simultánea.*

*El modelo de procesamiento que utilizaremos es el de concurrencia solapada (interleaved concurrency), que considera las siguientes hipótesis:*

- Disponemos de un único procesador que puede ejecutar múltiples transacciones simultáneamente.
- Cada transacción está formada por una secuencia de instrucciones atómicas, que el procesador ejecuta de a una a la vez.
- En cualquier momento el scheduler puede suspender la ejecución de una transacción, e iniciar o retomar la ejecución de otra.

*Si tuviéramos múltiples unidades de procesamiento, el modelo a utilizar sería el de procesamiento paralelo. Las herramientas teóricas que desarrollemos para concurrencia solapada, son en su mayoría extensibles al caso de procesamiento paralelo.*

*Consideremos que nuestra base de datos está formada por ítems. Un ítem puede representar:*

- El valor de un atributo en una fila determinada de una tabla.
- Una fila de una tabla.
- Un bloque del disco.
- Una tabla.

*Las instrucciones atómicas básicas de una transacción sobre la base de datos serán:*

- **leer\_item(x):** Lee el valor del ítem x, cargándolo en una variable en memoria
- **escribir\_item(x):** Ordena escribir el valor que está en memoria del ítem x en la base de datos

**Observación 1:** Desde ya, el código de la transacción contendrá instrucciones que involucran la manipulación de datos en memoria (por ejemplo, realizar la junta en memoria de dos tablas ya leídas), pero las mismas no afectan al análisis de concurrencia.

**Observación 2:** Ordenar escribir no es lo mismo que efectivamente escribir en el medio de almacenamiento persistente en que se encuentra la base de datos! El nuevo valor podría quedar temporalmente en un buffer en memoria.

## ACID

**Atomicidad:** Desde el punto de vista del usuario, las transacciones deben ejecutarse de manera atómica. Esto quiere decir que, o bien la transacción se realiza por completo, o bien no se realiza.

**Consistencia:** Cada ejecución, por si misma, debe preservar la consistencia de los datos. La consistencia se define a través de reglas de integridad: condiciones que deben verificarse sobre los datos en todo momento.

**Aislamiento:** El resultado de la ejecución concurrente de las transacciones debe ser el mismo que si las transacciones se ejecutaran en forma aislada una tras otra, es decir en forma serial. La ejecución concurrente debe entonces ser equivalente a alguna ejecución serial.

**Durabilidad:** Una vez que el SGBD informa que la transacción se ha completado, debe garantizarse la persistencia de la misma, independientemente de toda falla que pueda ocurrir.

Para garantizar las propiedades ACID, los SGBD disponen de mecanismos de recuperación que permiten deshacer/rehacer una transacción en caso de que se produzca un error o falla.

Para ello es necesario agregar a la secuencia de instrucciones de cada transacción algunas instrucciones especiales:

- **begin:** Indica el comienzo de la transacción.
- **commit:** Indica que la transacción ha terminado exitosamente, y se espera que su resultado haya sido efectivamente almacenado en forma persistente.
- **abort:** Indica que se produjo algún error o falla, y que por lo tanto todos los efectos de la transacción deben ser deshechos (rolled back).

Cuando se ejecutan transacciones en forma concurrente se da lugar a distintas situaciones anómalas que pueden violar las propiedades ACID.

La anomalía de **lectura sucia (dirty read)** se presenta cuando una transacción  $T_2$  lee un ítem que ha sido modificado por otra transacción  $T_1$  no commiteada.

Si luego  $T_1$  se deshace, la lectura que hizo  $T_2$  no es válida en el sentido de que la ejecución resultante puede no ser equivalente a una ejecución serial de las transacciones.

También se lo conoce con el nombre de "Temporary update" o "Read uncommitted data".

La anomalía de la **actualización perdida (lost update)** ocurre cuando una transacción modifica un ítem que fue leído anteriormente por una primera transacción que aún no commiteó.

En este caso, si la primera transacción luego modifica y escribe el ítem que leyó, el valor escrito por la segunda se perderá.

Si en cambio la primera transacción volviera a leer el ítem luego de que la segunda lo escribiera, se encontraría con un valor distinto. En este caso se lo conoce como lectura no repetible (unrepeatable read).

Ambas situaciones presentan un conflicto de tipo RW (read-write), seguido por otro de tipo WW o WR, respectivamente.

La anomalía de la **escritura sucia (dirty write)** ocurre cuando una transacción  $T_2$  escribe un ítem que ya había sido escrito por otra transacción  $T_1$  que luego se deshace.

El problema se dará si los mecanismos de recuperación vuelven al ítem a su valor inicial, deshaciendo la modificación realizada por  $T_2$ .

La anomalía del **fantasma (phantom)** se produce cuando una transacción  $T_1$  observa un conjunto de ítems que cumplen determinada condición, y luego dicho conjunto cambia porque algunos de sus ítems son modificados/creados/eliminados por otra transacción  $T_2$ .

Si esta modificación se hace mientras  $T_1$  aún se está ejecutando,  $T_1$  podría encontrarse con que el conjunto de ítems que cumplen la condición cambio.

Esta anomalía atenta contra la serializabilidad.

## Serializabilidad

Para analizar la serializabilidad de un conjunto de transacciones en nuestro modelo de concurrencia solapada, utilizaremos la siguiente notación breve para las instrucciones:

- $R_T(x)$ : La transacción  $T$  lee el ítem  $x$ .
- $W_T(x)$ : La transacción  $T$  escribe el ítem  $x$ .

- $b_T$ : Comienzo de la transaccion  $T$ .
- $c_T$ : Commit de la transacción  $T$ .
- $a_T$ : Abort de la transacción  $T$ .

Con esta notacion, podemos reescribir una transaccion general  $T$  como una lista de instrucciones, en donde  $m(T)$  representa la cantidad de instrucciones de  $T$ .

Un solapamiento entre dos transacciones  $T_1$  y  $T_2$  es una lista de  $m(T_1) + m(T_2)$  instrucciones, en donde cada instrucciones de  $T_1$  y  $T_2$  aparece una única vez, y las instrucciones de cada transaccion conservan el orden entre ellas dentro del solapamiento.

Dado un conjunto de transacciones  $T_1, T_2, \dots, T_n$  una ejecución serial es aquella en que las transacciones se ejecutan por completo una detrás de otra, en base a alguna orden  $T_{i,1}, T_{i,2}, \dots, T_{i,n}$ .

Decimos que un solapamiento de un conjunto de transacciones  $T_1, T_2, \dots, T_n$  es serializable cuando la ejecución de sus instrucciones en dicho orden deja a la base de datos en un estado equivalente a aquel en que la hubiera dejado alguna ejecución serial de  $T_1, T_2, \dots, T_n$ .

Nos interesa que los solapamiento producidos sean serializable, porque ellos garantizan la propiedad de aislamiento de las transacciones.

Pero, como evaluamos esta "equivalencia" entre ordenes de ejecución?

Deberíamos no solo mirar nuestra base de datos actual, que depende de un estado inicial particular anterior a la ejecución de las transacciones, sino pensar en cualquier estado inicial posible.

Existen entonces distintas nociones de equivalencia entre ordenes de ejecución de transacciones:

- **Equivalencia de resultados:** Cuando, dado un estado inicial particular, ambos ordenes de ejecución dejan a la base de datos en el mismo estado.
- **Equivalencia de conflictos:** Cuando ambos ordenes de ejecución poseen los mismos conflictos entre instrucciones. Esta noción es particularmente interesante porque no depende del estado inicial de la base de datos. Es la mas fuerte de las tres.
- **Equivalencia de vistas:** Cuando en cada orden de ejecución, cada lectura  $R_{T,i}(x)$  lee el valor escrito por la misma transaccion  $j$ ,  $W_{T,j}(x)$ . Además se pide que en ambos ordenes la ultima modificación de cada ítem  $x$  haya sido hecha por la misma transaccion.

Dado un orden de ejecución, un conflicto es un par de instrucciones  $(I_1, I_2)$  ejecutadas por dos transacciones distintas  $T_i$  y  $T_j$ , tales que  $I_2$  se encuentra mas tarde que  $I_1$  en el orden, y que responde a alguno de los siguiente esquemas:

- $R_{T,i}(x), W_{T,j}(x)$ : Una transaccion escribe un ítem que otra leyó.
- $W_{T,i}(x), R_{T,j}(x)$ : Una transaccion lee un ítem que otra escribió.
- $W_{T,i}(x), W_{T,j}(x)$ : Dos transacciones escriben un mismo ítem.

En otras palabras, tenemos un conflicto cuando dos transacciones distintas ejecutan instrucciones sobre un mismo ítem  $x$ , y al menos una de las dos instrucciones es una escritura.

Todo par de instrucciones consecutivas  $(I_1, I_2)$  de un solapamiento que no constituye un conflicto puede ser invertido en su ejecución (es decir, reemplazado por el par  $(I_2, I_1)$ ) obteniendo un solapamiento equivalente por conflictos al inicial.

### Ejemplo:

Indicar si el siguiente solapamiento de dos transacciones  $T_1$  y  $T_2$  es serializable por conflictos.

R_T_1(A);	W_T_1(A);	R_T_2(A);	W_T_2(A);	R_T_1(B);	W_T_1(B);	R_T_2(B);	W_T_2(B);
^	^	^	^	^	^	^	^
	+-----+	+-----+	+-----+		+-----+	+-----+	+-----+
+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+	+-----+

## Grafo de precedencias

La serializabilidad por conflictos puede ser evaluada a través del grafo de precedencias.

Dado un conjunto de transacciones  $T_1, T_2, \dots, T_n$  que acceden a determinados ítems  $X_1, X_2, \dots, X_p$ , el grafo de precedencias es un grafo dirigido simple que se construye de la siguiente forma:

- Se crea un nodo por cada transacción  $T_1, T_2, \dots, T_n$ .
- Se agrega un arco entre los nodos  $T_i$  y  $T_j$  con ( $i \neq j$ ) si y solo si existe algún conflicto de la forma  $(R_{T_i}(X_k), W_{T_j}(X_k))$ ,  $(W_{T_i}(X_k), R_{T_j}(X_k))$  o  $(W_{T_i}(X_k), W_{T_j}(X_k))$ .

Cada arco  $(T_i, T_j)$  en el grafo representa una precedencia entre  $T_i$  y  $T_j$ , e indica que para que el resultado sea equivalente por conflictos a una ejecución serial, entonces en dicha ejecución serial  $T_i$  debe preceder a (es decir, "ejecutarse antes que")  $T_j$ .

Opcionalmente podemos etiquetar el arco con el nombre del recurso que cause el conflicto.

Los conflictos son:

- $R_{T_i}(X); W_{T_j}(X)$
- $W_{T_i}(X); R_{T_j}(X)$
- $W_{T_i}(X); W_{T_j}(X)$

Un orden de ejecución es serializable por conflictos si y solo si su grafo de precedencias no tiene ciclos. Si un orden de ejecución es serializable por conflictos, el orden de ejecución serial equivalente puede ser calculado a partir del grafo de precedencias, utilizando el algoritmo de ordenamiento topológico.

## Control de concurrencia

El problema de control de concurrencia en vistas de garantizar el aislamiento admite dos enfoques:

- **Enfoque pesimista:** Busca garantizar que no se produzcan conflictos.
  - **Control de concurrencia basado en locks.**
- **Enfoque optimista:** Consiste en "dejar hacer" a las transacciones, y deshacer (rollback) una de ellas si en fase de validación se descubre un conflicto. Conveniente cuando la probabilidad de conflicto es baja.
  - **Control de concurrencia basado en timestamps.**
  - **Snapshot Isolation.**
  - **Control de concurrencia multiversión (MVCC).**

**Basado en locks.**

En este método, el SGBD utiliza locks para bloquear a los recursos (los ítems) y no permitir que más de una transacción los use en forma simultánea. Los locks son insertados por el SGBD como instrucciones especiales en medio de la transacción. Una vez insertados, las transacciones compiten entre ellas por su ejecución. Veremos que es posible -aunque no trivial- garantizar la serializabilidad utilizando locks.

Los locks o candados son variables asociadas a determinados recursos, y que permiten regular el acceso a los mismos en los sistemas concurrentes.

Son una herramienta para resolver el problema de exclusión mutua.

Un lock debe disponer de dos primitivas de uso, que permiten tomar y liberar el recurso X asociado al mismo:

- Acquire(X) o Lock(X) (L(X)).
- Release(X) o Unlock(X) (U(X)).

Tienen carácter bloqueante: Cuando una transacción tiene un lock sobre un ítem X, ninguna otra transacción puede adquirir un lock sobre el mismo ítem hasta tanto la primera no lo libere.

Es fundamental que dichas primitivas sean atómicas. Es decir, la ejecución de la primitiva Lock(X) sobre un recurso X no puede estar solapada con una ejecución semejante en otra transacción.

En general, los SGBD implementan locks de varios tipos. Los dos tipos de locks principales son:

- Locks de escritura o "de acceso exclusivo" (L\_EX(x)).
- Locks de lectura o "de acceso compartido" (L\_SH(X)).

Cuando una transacción posee un lock de acceso exclusivo sobre un ítem, ninguna otra transacción puede tener un lock de ningún tipo sobre ese mismo ítem.

Pero muchas transacciones pueden poseer locks de acceso compartido sobre un mismo ítem simultáneamente.

El empleo de locks por si solos no alcanza. Una transacción podría adquirir un lock sobre un ítem para leerlo, luego liberarlo, y mas tarde volver a adquirirlo para leerlo y modificarlo. Si en el intervalo otra transacción lo lee y escribe (aun tomando un lock), podría producirse la anomalía de la lectura no repetible.

El protocolo más comúnmente utilizado para adquisición y liberación de locks es el protocolo de lock de dos fases (2PL, two-phase lock).

#### **Protocolo de lock de dos fases (2PL).**

Una transacción no puede adquirir un lock luego de haber liberado un lock que había adquirido.

La regla divide naturalmente en dos fases a la ejecución de la transacción:

- Una fase de adquisición de locks, en la que la cantidad de locks adquiridos crece.
- Una fase de liberación de locks, en la que la cantidad de locks adquiridos decrece.

El cumplimiento de este protocolo es condición suficiente para garantizar que cualquier orden de ejecución de un conjunto de transacciones sea serializable.

Sin embargo, la utilización de locks introduce otros dos problemas potenciales que antes no teníamos:

- Bloqueo (deadlock).
- Inanición o postergación indefinida (livelock).

#### **Mecanismos de prevención de deadlocks:**

- Que cada transaccion adquiera todos los locks que necesita antes de comenzar su primera instrucción, y en forma simultanea. ( $Lock(X_q, X_w, \dots, X_n)$ ).
- Definir un ordenamiento de los recursos, y obligar a que luego todas las transacciones respeten dicho ordenamiento en la adquisición de locks.
- Métodos basados en timestamps.

La limitación del enfoque preventivo es que será necesario saber que recursos serán necesarios de antemano.

## Mecanismos de detección de deadlocks

- Analizar el **grafo de alocación de recursos**: un grafo dirigido que posee a las transacciones y los recursos como nodos, y en el cual se coloca un arco de una transaccion a un recurso cada vez que una transaccion espera por un recurso, y un arco de un recurso a una transaccion cada vez que la transaccion posee al lock de dicho recurso.
  - Cuando se detecta un ciclo en este grafo, se aborta (rollback) una de las transacciones involucradas.
  - El concepto es muy similar al del grafo de precedencias para un solapamiento.
- Definir un **timeout** para la adquisición del  $Lock(X)$ , después del cual se aborta la transaccion.

La **inanición** de una condición vinculada con el deadlock, y ocurre cuando una transaccion no logra ejecutarse por un periodo de tiempo indefinido.

Puede suceder por ejemplo, si ante la detección de un deadlock se elige siempre a la misma transaccion para ser abortada.

La solución mas común consistente en encolar los pedidos de locks, de manera que las transacciones que esperan desde hace mas tiempo por un recurso tengan prioridad en la adquisición de su lock.

Generalmente los SGBDs cuentan con estructuras de búsqueda de tipo árbol B+, tales que los bloques de datos se encuentran en las hojas.

A los locks que se aplican sobre los nodos de un índice se los denomina **index locks**.

Para mantener la serializabilidad en el acceso a estas estructuras, es necesario seguir las siguientes reglas:

- Todos los nodos accedidos deben ser lockeados.
- Cualquier nodo puede ser el primero en ser lockeado por la transaccion (aunque generalmente es la raíz).
- Cada nodo subsecuente puede ser lockeado solo si se posee un lock sobre su nodo padre.
- Los nodos pueden ser deslockeados en cualquier momento.
- Un nodo que fue deslockeado no puede volver a ser lockeado.

## Protocolo del cangrejo

A partir de las reglas anteriores podemos proponer el siguiente protocolo para el acceso concurrente a estructuras de árbol.

- Comenzar obteniendo un lock sobre el nodo raíz.
- Hasta llegar a el/los nodo/s deseado/s, adquirir un lock sobre el/los hijo/s que quiere acceder, y liberar el lock sobre el padre si los nodos hijo son seguros (es decir, el nodo hijo no esta lleno si estamos haciendo una inserción, ni esta justo por la mitad en el caso de una eliminación).
- Una vez terminada la operación, deslockear todos los nodos.

## Basado en timestamps

Se asigna a cada transaccion  $T_i$  un timestamp  $TS(T_i)$ . Los timestamps deben ser únicos, y determinaran el orden serial respecto al cual el solapamiento deberá ser equivalente.

Se permite la ocurrencia de conflictos, pero siempre que las transacciones de cada conflicto aparezcan de acuerdo al orden serial equivalente:

- $W_{T_i}(X), R_{T_j}(X) \rightarrow TS(T_i) < TS(T_j)$

Al no emplear locks, este método está exento de deadlocks.

Se debe mantener en todo instante, para cada ítem  $X$ , la siguiente información:

- **read\_ts(X):** ES el  $TS(T)$  correspondiente a la transaccion más joven -de mayor  $TS(T)$ - que leyó al ítem  $X$ .
- **write\_ts(X):** Es el  $TS(T)$  correspondiente a la transaccion más joven -de mayor  $TS(T)$ - que escribió el ítem  $X$ .

#### Lógica de funcionamiento:

- Cuando una transaccion  $T_i$  quiere ejecutar un  $R(X)$ :
  - Si una transaccion posterior  $T_j$  modifico el ítem,  $T_i$  deberá ser abortada (read too late).
  - De lo contrario, actualiza  $read\_ts(X)$  y lee.
- Cuando una transaccion  $T_i$  quiere ejecutar un  $W(X)$ :
  - Si una transaccion posterior  $T_j$  leyó o escribió el ítem,  $T_i$  deberá ser abortada (write too late).
  - De lo contrario, actualiza  $write\_ts(X)$  y escribe.

Observación: La ejecución de los  $R(X)$  y  $W(X)$ , y actualización de los  $read\_ts(X)$  y  $write\_ts(X)$  se centraliza en el scheduler.

La lógica en el caso de ejecutar una escritura puede ser mejorada, utilizando la regla conocida como Thomas Write Rule:

Si cuando  $T_i$  intenta escribir un ítem encuentra que una transaccion posterior  $T_j$  ya lo escribió, entonces  $T_i$  puede descartar su actualización sin riesgos, siempre y cuando el ítem no haya sido leído por ninguna transaccion posterior a  $T_i$ .

Entonces: Si  $read\_ts(X) \leq ts(T_i)$ , no hacemos ninguna escritura y no modificamos nada. Si en cambio  $read\_ts(X) > ts(T_i)$ , definitivamente debemos abortar  $T_i$ .

Al utilizar esta mejora no queda garantizada la serializabilidad por conflictos, pero si la serializabilidad por vistas.

#### Snapshot Isolation

En este método, cada transaccion ve una Snapshot de la base de datos correspondiente al instante de su inicio.

**Ventajas:** Permite un mayor solapamiento, ya que lecturas que hubieran sido bloqueadas utilizando locks, ahora siempre puedan realizarse.

**Desventajas:** Requiere mayor espacio en disco o memoria, al tener que mantener múltiples versiones de los mismos ítems. Cuando ocurren conflictos de tipo WW entre transacciones, obliga a deshacer una de ellas.

Cuando dos transaccion intentan modificar un mismo ítem de datos, generalmente gana aquella que hace primero su commit, mientras que la otra deberá ser abortada (first-committer-wins).

Esto por si solo no alcanza para garantizar la serializabilidad. Debe combinarse con dos elementos mas:

- Validación permanente con el grafo de precedencias buscando ciclos de conflictos RW.

- Locks de predicados en el proceso de detección de conflictos, para detectar precedencias.

### Solución de anomalía del fantasma

- Bajo control de concurrencia basado en locks:
  - Locks de tablas: cuando se produce un  $R(X \mid cond)$ , bloquear el acceso a toda la tabla a la que  $X$  pertenece. Es una solución casi trivial, y poco eficiente.
  - Locks de predicados: bloquear todas aquellas tuplas que podrían cumplir la condición, evitando incluso futuras inserciones de tuplas que también la cumplan.
- Bajo Snapshot Isolation
  - No puede ocurrir, porque la transacción ve la misma snapshot a lo largo de su vida.
- Bajo control de concurrencia basado en timestamps
  - Utilizar índices de tipo árbol, y mantener registros `read_ts(l)` y `write_ts(l)` también para los nodos del árbol.

## Niveles de aislamiento

SQL permite definir el nivel de aislamiento de las transacciones con el comando `SET TRANSACTION ISOLATION LEVEL`.

```
SET TRANSACTION ISOLATION LEVEL
READ UNCOMMITTED | READ COMMITTED | REPEATABLE READ | SERIALIZABLE;
```

Y para definir una transacción:

```
START TRANSACTION [ISOLATION LEVEL ...]
  (# sql commands
  )
  COMMIT | ROLLBACK
```

Distintas motores pueden implementar los niveles de aislamiento con distintas técnicas, pero de acuerdo con el nivel de aislamiento configurado, el SGBD debe garantizar que ciertas anomalías no ocurran.

De acuerdo con el nivel de aislamiento elegido, pueden producirse o no anomalías.

Nivel de aislamiento	Lectura sucia	Lectura no repetible	Fantasma
READ UNCOMMITTED	✗	✗	✗
READ COMMITTED	✓	✗	✗
REPEATABLE READ	✓	✓	✗
SERIALIZABLE	✓	✓	✓

El tick representa que tal nivel de aislamiento impide alguna de las anomalías.

Si bien la anomalía de "escritura sucia" (dirty write) no se menciona en el estándar, la misma debería ser proscripta en todos los niveles de aislamiento, y prácticamente todos los SGBD lo hacen.

## Recuperabilidad

La serializabilidad de las transacciones ya nos asegura la propiedad de aislamiento. Nos interesa ahora asegurar que una vez que se commitea la transacción, esta perdure en disco.

**Definición:** Un solapamiento es recuperable si y solo si ninguna transacción  $T$  realiza el commit hasta tanto todas las transacciones que escribieron datos antes de que  $T$  los leyera hayan commiteado.

Dado un solapamiento recuperable, puede ser necesario deshacer (abortar) una transacción antes de llegar su commit, y para ello el SGBD deberá contar con información que es almacenada por su gestor de recuperación en un log (bitácora).

Un SGBD no debería jamás permitir la ejecución de un solapamiento que no sea recuperable.

¿Cómo deshacemos los efectos de una transacción  $T_j$  que hay que abortar, sin afectar la serializabilidad de las transacciones restantes?

Si las modificaciones por  $T_j$  no fueron leídas por nadie, entonces basta con procesar el log de  $T_j$  en forma inversa para deshacer sus efectos (rollback).

Pero si una transacción  $T_i$  leyó un dato modificado por  $T_j$ , entonces será necesario hacer el rollback de  $T_i$  para volverla a ejecutar. Sería conveniente que  $T_i$  no hubiera commiteado aún!

**Resultado:** Si un solapamiento de transacciones es recuperable, entonces nunca será necesario deshacer transacciones que ya hayan commiteado.

Puede producir una cascada de rollback.

Que un solapamiento sea recuperable, no implica que no sea necesario tener que hacer rollbacks en cascada de transacciones que aún no commitearon.

Para evitar los rollbacks en cascada es necesario que una transacción no lea valores que aún no fueron commiteados. Esto es más fuerte que la condición de recuperabilidad.

#### **Protocolo de lock de dos fases estricto (S2PL)**

Los locks también pueden ayudar a asegurar la recuperabilidad. El protocolo de 2PL estricto (S2PL) emplea la siguiente regla:

Una transacción no puede adquirir un lock luego de haber liberado un lock que había adquirido, y además los locks de escritura sólo pueden ser liberados después de haber commiteado la transacción.

#### **Protocolo de lock de dos fases riguroso (R2PL)**

En caso de no diferenciar tipos de lock, se convierte en riguroso (R2PL):

Los locks sólo pueden ser liberados después del commit.

Garantizan que todo solapamiento sea no sólo serializable, sino también recuperable, y que no se producirán cascadas de rollbacks al deshacer una transacción.

#### **Control de concurrencia basado en timestamps**

En este método, cuando se aborta una transacción  $T_i$ , cualquier transacción que haya usado datos que  $T_i$  modificó debe ser abortada en cascada.

- Para garantizar la recuperabilidad se puede escoger entre varias opciones:
  - (a) No hacer el *commit* de una transacción hasta que todas aquellas transacciones que modificaron datos que ella leyó hayan hecho su *commit*. Esto garantiza recuperabilidad.
  - (b) Bloquear a la transacción lectora hasta tanto la escritora haya hecho su *commit*. Esto evita rollbacks en cascada.
  - (c) Hacer todas las escrituras durante el *commit*, manteniendo una copia paralela de cada ítem para cada transacción. Para esto, la escritura de los ítems en el *commit* deberá estar centralizada y ser atómica.

## ▼ IX - Recuperación

### Fallas

Los sistemas reales sufren múltiples tipos de fallas:

- Fallas de sistema: Por errores de software ó hardware que detiene la ejecución de un programa: fallas de segmentación, división por cero, fallas de memoria.
- Fallas de aplicación: Aquellas que provienen desde la aplicación que utiliza la base de datos. Por ejemplo, la cancelación o vuelta atrás de una transacción.
- Fallas de dispositivos: Aquellas que provienen en un daño físico en dispositivos como discos rígidos o memoria.
- Fallas naturales externas: Son aquellas que provienen desde afuera del hardware en que se ejecuta nuestro SGBD. Ejemplos: Caídas de tensión, terremotos, incendios, ...

En situaciones catastróficas como 3 ó 4, es necesario contar con mecanismos de backup para recuperar la información.

### Actualización

**Inmediata:** Se guardan a disco inmediatamente.

**Diferido:** Se guardan en disco después del *commit*.

### Gestor de Recuperación

#### Estructura de log

- El log almacena generalmente los siguientes registros:
  - (**BEGIN**,  $T_{id}$ ): Indica que la transacción  $T_{id}$  comenzó.
  - (**WRITE**,  $T_{id}$ ,  $X$ ,  $x_{old}$ ,  $x_{new}$ ): Indica que la transacción  $T_{id}$  escribió el ítem  $X$ , cambiando su viejo valor  $x_{old}$  por un nuevo valor  $x_{new}$ .
  - (**READ**,  $T_{id}$ ,  $X$ ): Indica que la transacción  $T_{id}$  leyó el ítem  $X$ .
  - (**COMMIT**,  $T_{id}$ ): Indica que la transacción  $T_{id}$  commitió.
  - (**ABORT**,  $T_{id}$ ): Indica que la transacción  $T_{id}$  abortó.

En realidad, según el algoritmo de recuperación que utilicemos, no siempre será necesario guardar los valores anteriores y actuales en el WRITE, y no siempre será necesario guardar los READs.

#### Reglas WAL y FLC

- WAL (Write Ahead Log)
- FLC (Force Log at Commit)

La **regla WAL** indica que antes de guardar un ítem modificado en disco, se debe escribir el registro de log correspondiente, en disco.

La **regla FLC** indica que antes de realizar el commit el log debe ser volcado a disco.

## Algoritmos de Recuperación

En los tres se asume que los solapamientos de transacciones son:

- Recuperables
- Evitan rollbacks en cascada

## Algoritmo UNDO

Antes de que una modificación sobre un ítem  $X$  por parte de una transacción no commiteada sea guardada en disco (flush), se debe salvaguardar en el log en disco el último valor commiteado de ese ítem.

- Para cumplir con la regla se utiliza el siguiente procedimiento:
  - 1 Cuando una transacción  $T_i$  modifica el ítem  $X$  remplazando un valor  $v_{old}$  por  $v$ , se escribe (**WRITE**,  $T_i$ ,  $X$ ,  $v_{old}$ ) en el log, y se hace flush del log a disco.
  - 2 El registro (**WRITE**,  $T_i$ ,  $X$ ,  $v_{old}$ ) debe ser escrito en el log en disco (flushed) antes de escribir (flush) el nuevo valor de  $X$  en disco (WAL).
  - 3 Todo ítem modificado debe ser guardado en disco antes de hacer commit.
  - 4 Cuando  $T_i$  hace **commit**, se escribe (**COMMIT**,  $T_i$ ) en el log y se hace flush del log a disco (FLC).

*Los tres primeros puntos aseguran que todas las modificaciones realizadas sean escritas a disco antes de que la transacción termine.*

*De esta forma, una vez cumplido el paso 4, ya nunca será necesario hacer REDO. Si la transacción falla antes ó durante el punto 4, será deshecha (UNDO) al reiniciar.*

*Cuando el sistema reinicia se siguen los siguientes pasos:*

- Se recorre el log de adelante hacia atrás, y por cada transacción de la que no se encuentra el COMMIT se aplica cada uno de los WRITE para restaurar el valor anterior a la misma en disco.
- Luego, por cada transacción de la que no se encontró el COMMIT se escribe (ABORT, T) en el log y se hace flush del log a disco.

*Obsérvese que también podría ocurrir una falla durante el reinicio. Esto no es un problema porque el procedimiento de reinicio es idempotente: Si se ejecuta más de una vez, no cambiará el resultado.*

## Algoritmo REDO

Antes de realizar el commit, todo nuevo valor v asignado por la transacción debe ser salvaguardado en el log, en disco.

*El ítem es actualizado en disco luego de commitear la transacción.*

■ Para cumplir con la regla se utiliza el siguiente procedimiento:

- 1 Cuando una transacción  $T_i$  modifica el ítem X remplazando un valor  $v_{old}$  por  $v$ , se escribe (WRITE,  $T_i$ , X,  $v$ ) en el log.
- 2 Cuando  $T_i$  hace commit, se escribe (COMMIT,  $T_i$ ) en el log y se hace flush del log a disco (FLC). Recién entonces se escribe el nuevo valor en disco.

- En el algoritmo REDO, una transacción debe commitear sin haber guardado en disco todos sus ítems modificados.
- Ante una falla posterior al commit, entonces, será necesario reescribir (REDO) TODOS LOS VALORES QUE LA TRANSACCIÓN HABÍA ASIGNADO A LOS ÍTEMES.
- Esto implicará recorrer todo el log de atrás para adelante aplicando cada caso el WRITE.

■ Cuando el sistema reinicia se siguen los siguientes pasos:

- 1 Se analiza cuáles son las transacciones de las que está registrado el COMMIT.
- 2 Se recorre el log de atrás hacia adelante volviendo a aplicar cada uno de los WRITE de las transacciones que commitearon, para asegurar que quede actualizado el valor de cada ítem.
- 3 Luego, por cada transacción de la que no se encontró el COMMIT se escribe (ABORT, T) en el log y se hace flush del log a disco.

Entre UNDO y REDO, el log es prácticamente el mismo con la salvedad de que UNDO registra los valores viejos a los cambios mientras que REDO los nuevos. Al momento de recuperar la base, UNDO recorre el log de más nuevo a más viejo, mientras que REDO al revés.

## Algoritmo UNDO/REDO

En este algoritmo es necesario cumplir con ambas reglas a la vez. El procedimiento es el siguiente:

- Cuando una transacción  $T_i$  modifica el ítem  $X$  remplaza su valor viejo por el nuevo, se guardan ambos.
- El registro debe ser escrito en el log en disco antes de escribir el nuevo valor de  $X$  en disco.
- Cuando  $T_i$  hace commit, se escribe en el log y se hace flush del log a disco.
- Los ítems modificados pueden ser guardados en disco antes o después de hacer commit.

### ■ Cuando el sistema reinicia se siguen los siguientes pasos:

- 1 Se recorre el log de adelante hacia atrás, y por cada transacción de la que no se encuentra el COMMIT se aplica cada uno de los WRITE para restaurar el valor anterior a la misma en disco. U
- 2 Luego se recorre de atrás hacia adelante volviendo a aplicar cada uno de los WRITE de las transacciones que commitearon, para asegurar que quede asignado el nuevo valor de cada ítem.
- 3 Finalmente, por cada transacción de la que no se encontró el COMMIT se escribe (ABORT,  $T$ ) en el log y se hace flush del log a disco.

## Puntos de Control

Cuando reiniciamos el sistema no sabemos hasta donde tenemos que retroceder en el archivo de log. Aunque muchas transacciones antiguas ya commiteadas seguramente tendrán sus datos guardados ya en disco.

Para evitar este retroceso hasta el inicio del sistema y el crecimiento ilimitado de los archivos de log se utilizan puntos de control (checkpoints).

Un punto de control es un registro especial en el archivo de log que indica que todos los ítems modificados hasta ese punto han sido almacenados en disco.

La presencia de un checkpoint en el log implica que todas las transacciones que vinieron antes ya fueron guardadas a disco.

### Checkpoint inactivo (CKPT)

La creación de un checkpoint inactivo en el log implica la suspensión momentánea de todas las transacciones para hacer el flush de todos los buffers en memoria al disco. Donde se dejan de aceptar nuevas transacciones y se espera a que terminen las actuales.

- En el algoritmo UNDO, el procedimiento de *checkpointing inactivo* se realiza de la siguiente manera:
  - 1 Dejar de aceptar nuevas transacciones.
  - 2 Esperar a que todas las transacciones hagan su *commit* (es decir, escriban su registro de COMMIT en el *log* y lo vuelquen a disco).
  - 3 Escribir (CKPT) en el *log* y volcarlo a disco.
- Si el sistema cae justo después de escribir (CKPT) en el *log*, ¿es posible que alguno de los ítems modificados por alguna transacción no hayan sido guardados a disco?
  - No, porque en el algoritmo UNDO la presencia del registro de COMMIT en el *log* implica que todos los ítems fueron ya salvaguardados en disco.
- Durante la recuperación, sólo debemos deshacer las transacciones que no hayan hecho *commit*, hasta el momento en que encontrremos un registro de tipo (CKPT). De hecho, todo el archivo de *log* anterior al *checkpoint* podía ser eliminado.

#### Checkpoint activo (BEGIN CKPT, END CKPT)

Para aminorar la pérdida de tiempo de ejecución en el volcado a disco puede utilizarse una técnica conocida como *checkpointing activo*, que utiliza dos tipos de registros de checkpoint en donde  $t_{act}$  es un listado de todas las transacciones que se encuentran activas (es decir, que aún no hicieron *commit*). El procedimiento varía según cada algoritmo de recuperación.

- En la versión activa para el algoritmo UNDO, el procedimiento es el siguiente:
  - 1 Escribir un registro (BEGIN CKPT,  $t_{act}$ ) con el listado de todas las transacciones activas hasta el momento.
  - 2 Esperar a que todas esas transacciones activas hagan su *commit* (sin dejar por eso de recibir nuevas transacciones)
  - 3 Escribir (END CKPT) en el *log* y volcarlo a disco.
- En la recuperación, al hacer el *rollback* se dan dos situaciones:
  - Que encontrremos primero un registro (END CKPT). En ese caso, sólo debemos retroceder hasta el (BEGIN CKPT) durante el *rollback*, porque ninguna transacción incompleta puede haber comenzado antes).
  - Que encontrremos primero un registro (BEGIN CKPT). Esto implica que el sistema cayó sin asegurar los *commits* del listado de transacciones. Deberemos volver hacia atrás, pero sólo hasta el inicio de la más antigua del listado.

- En el algoritmo REDO con *checkpointing* activo se procede de la siguiente forma:
  - 1 Escribir un registro (BEGIN CKPT,  $t_{act}$ ) con el listado de todas las transacciones activas hasta el momento y volcar el log a disco.
  - 2 Hacer el volcado a disco de todos los ítems que hayan sido modificados por transacciones que ya commitearon.
  - 3 Escribir (END CKPT) en el *log* y volcarlo a disco.
- En la recuperación hay nuevamente dos situaciones:
  - Que encontremos primero un registro (END CKPT). En ese caso, deberemos retroceder hasta el (BEGIN ,  $T_x$ ) más antiguo del listado que figure en el (BEGIN CKPT) para rehacer todas las transacciones que commitearon. Escribir (ABORT,  $T_y$ ) para aquellas que no hayan commiteado.
  - Que encontremos primero un registro (BEGIN CKPT). Si el *checkpoint* llegó sólo hasta este punto no nos sirve, y entonces deberemos ir a buscar un *checkpoint* anterior en el *log*.
  
- En el algoritmo UNDO/REDO con *checkpointing* activo el procedimiento es:
  - 1 Escribir un registro (BEGIN CKPT,  $t_{act}$ ) con el listado de todas las transacciones activas hasta el momento y volcar el log a disco.
  - 2 Hacer el volcado a disco de todos los ítems que hayan sido modificados antes del (BEGIN CKPT).
  - 3 Escribir (END CKPT) en el *log* y volcarlo a disco.
- En la recuperación es posible que debamos retroceder hasta el inicio de la transacción más antigua en el listado de transacciones, para deshacerla en caso de que no haya commiteado, o para rehacer sus operaciones posteriores al BEGIN CKPT, en caso de que haya commiteado.
  
- En resumen:
  - En el algoritmo UNDO, escribimos el (END CKPT) cuando todas las transacciones del listado de transacciones activas hayan hecho *commit*.
  - Para el algoritmo REDO, escribimos (END CKPT) cuando todos los ítems modificados por transacciones que ya habían commiteado al momento del (BEGIN CKPT) hayan sido salvaguardados en disco.
  - En el UNDO/REDO escribimos (END CKPT) cuando todos los ítems modificados antes del (BEGIN CKPT) hayan sido guardados en disco.

## ▼ X - NoSQL

### Bases de Datos Distribuidas

Las bases de datos NoSQL buscan aumentar la velocidad de procesamiento y la capacidad de almacenar información, explotando las ventajas que brindan las redes de computadores y en particular Internet. Para ello implementan la funcionalidad de un sistema de gestión de bases de datos distribuido.

## Fragmentación

La **fragmentación** es la tarea de dividir un conjunto de **agregados** entre un conjunto de nodos. Se realiza con dos objetivos:

- Almacenar conjuntos muy grandes de datos que de lo contrario no podrían caber en un único nodo.
- Paralelizar el procesamiento, permitiendo que cada nodo ejecute una parte de las consultas para luego integrar los resultados.

Según la manera de fragmentar, podemos distinguir entre:

- Fragmentación horizontal: Los agregados se reparte entre los nodos, de manera que cada nodo almacena un subconjunto de agregados. Generalmente se asigna el nodo a partir del valor de alguno de los atributos del agregado.
- Fragmentación vertical: Distintos nodos guardan un subconjunto de atributos de cada agregado. Todos suelen compartir los atributos que conforman la clave.

Muchas veces se utiliza una combinación de ambas.

## Replicación

La **replicación** es el proceso por el cual se almacenan múltiples copias de un mismo dato en distintos nodos del sistema. Nos brinda varias ventajas:

- Es un mecanismo de backup: permite recuperar el sistema en caso de fallas de disco o catastróficas.
- Permite repartir la carga de procesamiento si permitimos que las réplicas respondan consultas o actualizaciones.
- Garantiza cierta disponibilidad del sistema aún si se caen algunos nodos.

Cuando las réplicas sólo funcionan como mecanismo de backup se denominan réplicas secundarias. Cuando también pueden hacer procesamiento, se las conoce como réplicas primarias.

La replicación nos genera un nuevo problema a resolver: La consistencias de los datos. Puede darse la situación de que distintas réplicas almacenen (al menos, temporalmente) distintos valores para un mismo dato.

## Bases de Datos Clave-Valor

Estas bases de datos almacenan vectores asociativos ó diccionarios, es decir conjuntos formados por pares de elementos de forma (clave, valor).

Las claves son únicas (es decir, no puede haber dos pares que posean la misma clave), y el único requisito sobre su dominio es que sea comparable por igual (=).

Este tipo de bases de datos tienen cuatro operaciones elementales:

- Insertar un nuevo par (put)
- Eliminar un par existente (delete)
- Actualizar el valor de un par (update)

- Encontrar un par asociado a una clave particular (*get*)

Sus grandes ventajas son su **simplicidad, velocidad y escalabilidad**.

## Dynamo

*Es un key-store de Amazon.*

Está diseñado siguiendo una arquitectura orientada a servicio: la base de datos está distribuida en un server cluster que posee servidores web, routers de agregación y nodos de procesamiento (Dynamo instances).

Utiliza un método de lookup denominado **hashing consistente** que reduce la cantidad de movimientos de pares necesarios cuando cambia la cantidad de nodos S. Esto hace que sea muy sencillo agregar nodos en forma dinámica, con un impacto mínimo.

Utiliza un modelo de consistencia denominado consistencia eventual, que tolera pequeñas inconsistencias en los valores almacenados en distintas réplicas.

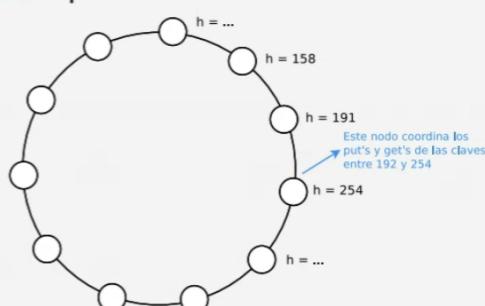
*Es totalmente descentralizado. Los nodos son peers entre sí. Carece de un punto único de falla.*

Disponemos de una función de hash  $h$  que, dada una clave  $k$ , devuelve un valor  $h(k)$  entre 0 y  $2^M - 1$ , en donde  $M$  representa la cantidad de bits del resultado.

*El valor de la función de hash para un par dado es el que determina en cuál de los S nodos el mismo será almacenado. Esto es lo que se conoce como una tabla de hash distribuida (DHT).*

A diferencia de otras DHTs en que el nodo asignado se determina como  $h(k) \bmod S$ , en Dynamo se utiliza una técnica ligeramente distinta, conocida como hashing consistente.

- Al identificador de cada nodo de procesamiento (generalmente, su dirección IP) se le aplica la misma función de hash. A partir de los hashes, los nodos se organizan virtualmente en una estructura de anillo por *hash* creciente.



- **Regla:** Un par  $(k, v)$  se replicará en los  $N$  servidores siguientes a  $h(k)$ , que conformarán el *listado de preferencia* para esa clave.

## Modelos de consistencia

En el estudio de la replicación en las bases de datos distribuidas se utilizan distintos modelos de consistencia.

Uno de los modelos más fuertes, y que proviene de las bases de datos centralizadas, es el de consistencia secuencial.

Partimos de una serie de procesos que ejecutan instrucciones de lectura,  $R_{P\_i}(X)$  y de escritura,  $W_{P\_i}(X)$ , sobre una base de datos distribuida.

### Consistencia Secuencial

Se dice que una base de datos distribuida tiene **consistencia secuencial** cuando "el resultado de cualquier ejecución concurrente de los procesos es equivalente al de alguna ejecución secuencial en la que las instrucciones de los procesos se ejecutan una después de otra".

### Consistencia Causal

En este modelo se busca capturar eventos que pueden estar causalmente relacionados. Si un evento  $b$  fue influenciado por un evento  $a$ , la causalidad requiere que todos vean al evento  $a$  antes que al evento  $b$ .

**Ejemplo:** Supongamos que un proceso  $P_1$  escribe un ítem  $X$ . Simultáneamente, un proceso  $P_2$  lee un ítem  $X$  y escribe el ítem  $Y$ . Las dos escrituras están causalmente relacionadas, porque operan sobre el mismo ítem. Entonces, el modelo requiere que todos las vean en el mismo orden.

Dos eventos que no están causalmente correlacionados se dicen concurrentes, y no es necesario que sean vistos por todos en el mismo orden.

En el modelo de consistencia causal, "dos escrituras que están potencialmente causalmente relacionadas deben ser vistas por todos en el mismo orden".

### Consistencia Eventual

Este modelo está basado en la siguiente observación: En la mayoría de los sistemas reales, son pocos los procesos que realizan modificaciones o escrituras, mientras que la mayor parte sólo lee. ¿Qué rápido necesitamos que las actualizaciones de un proceso que escribe sean vistas por los procesos que leen?

Estas situaciones pueden tolerar un grado bastante alto de inconsistencia.

Decimos entonces que una ejecución tiene consistencia eventual cuando "si en el sistema no se producen modificaciones por un tiempo suficientemente grande, entonces eventualmente todos los procesos verán los mismos valores".

En otras palabras, esto implica que eventualmente todas las réplicas llegarán a ser consistentes (guardarán los mismos valores).

Dynamo provee un modelo de consistencia eventual, que permite que las actualizaciones se propaguen a las réplicas de forma asíncrona.

Gracias a esto, las lecturas y escrituras pueden devolver el control rápidamente.

Cuando un nodo recibe un put sobre una clave, no necesita propagarlo a las  $N - 1$  réplicas antes de confirmar la escritura. Dado que las operaciones get pueden realizarse sobre cualquier réplica, es posible leer un valor no actualizado.

Se definen dos parámetros adicionales:

- $W \leq N$ : Quorum de escritura
- $R \leq N$ : Quorum de lectura

Un nodo puede devolver un resultado de escritura exitosa luego de recibir la confirmación de escritura de otros  $W - 1$  nodos del listado de preferencia.

Un nodo puede devolver el valor de una clave leída luego de disponer de la lectura de  $R$  nodos distintos (incluido él mismo). Valores mayores de  $R$  brindan tolerancia a fallas como corrupción de datos ó ataques externos, pero hacen más lenta la lectura.

■ Algunos valores comunes de  $R$  y  $W$  son:

N	R	W	
3	2	2	Buena durabilidad y latencia (paper).
3	3	1	Lectura más lenta, pero pobre durabilidad. Escritura rápida.
3	1	3	Escripturas más lentas. Muy buena durabilidad y lectura rápida.

■ Para mantener sincronizadas las réplicas, Dynamo utiliza una estructura llamada **Merkle tree** que consiste en un árbol en que cada nodo no-hoja es un *hash criptográfico* de los valores de sus hijos.

## Bases de Datos orientadas a Documentos

Un documento es una unidad estructural de información (agregado), que almacena datos bajo una cierta estructura.

Sin necesidad de definir un esquema rígido para la estructura del documento, estas bases de datos ofrecen la posibilidad de manejar estructuras un poco más complejas que un simple par (clave, valor).

Generalmente, un documento se define como un conjunto de pares (clave, valor).

### MongoDB

- Basada en hashes para identificar a los objetos.
- No utiliza esquemas (schema-free). No existe un DDL.

- Los documentos tienen formato BSON.
- Almacena por (clave, valor).
- Desarrollada en C++. Tiene APIs en múltiples lenguajes: Python, Java, C#.
- Su implementación de la operación de junta es limitada.
- Organiza los datos de una base de datos en colecciones que contienen documentos.

Modelo relacional	MongoDB
Esquema	Base de datos
Relación	Colección $\rightarrow$ "tabla"
$\leftarrow$ fila Tupla	Documento
$\nwarrow$ Atributo	Campo $\rightarrow$ "clave"

Los documentos dentro de una colección se identifican a través de un campo "\_id" (clave primaria).

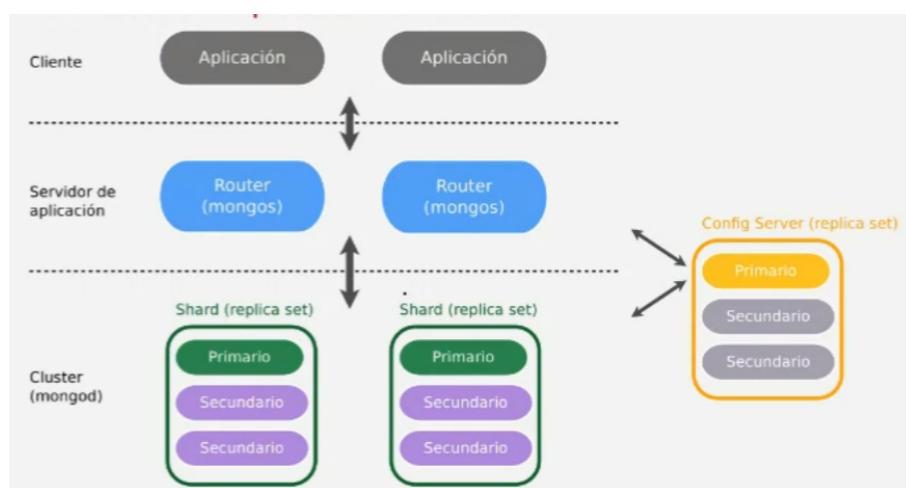
Si no lo indicamos, MongoDB le asigna un hash de 12 bytes.

El particionado de las colecciones se realiza a partir de un shard key., La shard key es un atributo ó conjunto de atributos de la colección que se escoge al momento de construir el sharded cluster.

La asignación de documentos a shards se hace dividiendo en rangos los valores de la shard key (range-based sharding), o bien a partir de una función de hash aplicada sobre su valor (hashed sharding).

El **sharding** permite disminuir el tiempo de respuesta en sistemas de alta carga de consultas, al distribuir el trabajo de procesamiento entre varios nodos.

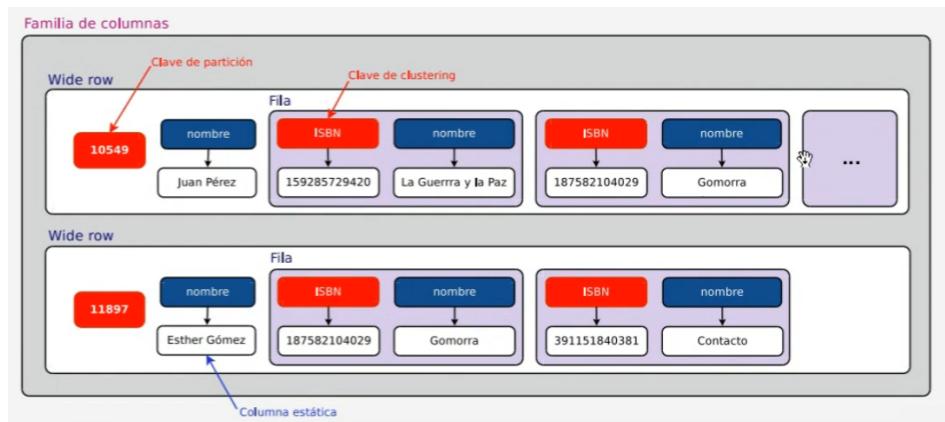
Mongo también brinda tolerancia a fallas a través de un mecanismo de replicación de shards.



El esquema de réplicas es de master-slave with automated failover:

- Cada shard pasa a tener un servidor mongod principal (master), y uno o más servidores mongod secundarios (slaves). El conjunto de réplicas de un shard se denomina replica set.

## Bases de Datos Wide Column



Son una evolución de las bases de datos clave/valor, ya que agrupan los pares vinculados a una misma entidad como columnas asociadas a una misma clave primaria.

## Cassandra

No es estrictamente orientada a columnas, no es libre de esquema, arquitectura share-nothing.

El concepto análogo de la tabla es el column family, una fila está formada por:

- Una clave compuesta (atributo ó conjunto de atributos)
- Un conjunto de pares clave-valor ó columnas

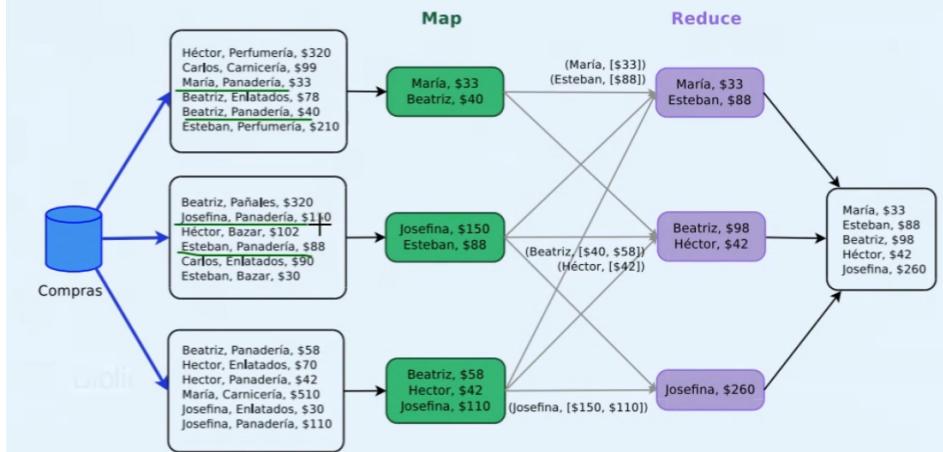
## El modelo MapReduce

Es una técnica que brinda un marco flexible para el procesamiento paralelo de grandes volúmenes de datos. Fue propuesto por Google en un paper de 2004, luego tomado por Yahoo! para la implementación de Hadoop (2006).

Sin embargo, MapReduce no introdujo ningún concepto nuevo. Es una reimplementación de conceptos de agregación que ya eran conocidos 30 años atrás.

El concepto detrás de MapReduce es el de dividir una entrada de datos en porciones que puedan ser ejecutadas por distintas unidades de procesamiento, y luego integrar el resultado para generar una salida, que a su vez servirá de entrada a otra etapa de procesamiento.

- En el siguiente ejemplo, queremos calcular cuál fue el gasto mensual de cada cliente en el rubro “Panadería”.



- Hadoop es la implementación de MapReduce de Yahoo!.
- Está programado en Java, y almacena sus datos en un sistema de archivos *ad hoc* denominado HDFS (*Hadoop Distributed File System*).
- Hadoop escribe todos los datos de las etapas intermedias en disco. (*materialización intermedia*)
- Cada etapa del *pipeline* MapReduce lee desde la entrada estándar (*stdin*) y produce sus resultados en la salida estándar (*stdout*). Cuando escribimos un *pipeline*, podemos testearlo ejecutando:

**Ejercicio**

```
cat entrada.txt | python mapper.py | sort -k1,1 |
python reducer.py > salida.txt
```

## Teorema CAP

En 1998 el científico E. Brewer postuló la imposibilidad de que un sistema de bases de datos distribuido garantice simultáneamente el máximo nivel de:

- (C) *Consistencia*
- (A) *Disponibilidad*
- (P) *Tolerancia a particiones*

En 2002 Seth Gilbert y Nancy Lynch presentaron una prueba formal del resultado.

La **consistencia** es la propiedad de que en un instante determinado el sistema muestre un único valor de cada ítem de datos a los usuarios. Su nivel máximo es la *consistencia secuencial*, en la que todas las operaciones de lectura/escritura distribuidas en el sistema pueden ordenarse de forma tal que toda lectura de un ítem siempre lea el último valor escrito en ese ítem. Alcanzar consistencia secuencial requiere de un alto nivel de sincronización entre los nodos.

La **disponibilidad** consiste en que toda consulta que llega a un nodo del sistema distribuido que no está caído reciba una respuesta efectiva (es decir, sin errores).

La **tolerancia a particiones** consiste en que el sistema pueda responder una consulta aún cuando algunas conexiones entre algunos pares de nodos estén caídas.

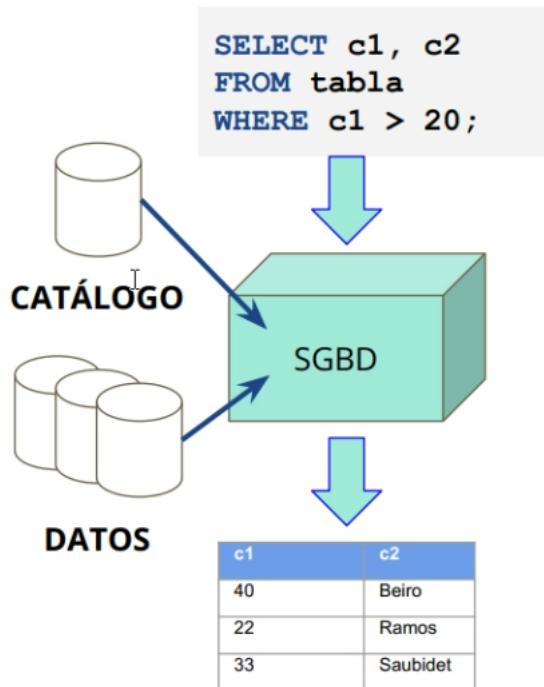
- El Teorema CAP dice entonces que a lo sumo podremos ofrecer 2 de las 3 garantías:
  - **AP:** Si la red está particionada, podemos optar por seguir respondiendo consultas aún cuando algunos nodos no respondan. Garantizaremos disponibilidad, pero el nivel de consistencia no será el máximo.
  - **CP:** Con la red particionada, si queremos garantizar consistencia máxima no podremos garantizar disponibilidad. Es posible que no podamos responder una consulta en forma efectiva porque esperamos mensajes de confirmación desde nodos que no pueden comunicarse.
  - **CA:** Si queremos consistencia y disponibilidad, entonces no podremos tolerar que una cantidad indeterminada de enlaces se caiga.

## Propiedades BASE

- Las propiedades BASE representan un sistema distribuido con:
  - **(B)** Disponibilidad básica (*basic availability*): El SGBD distribuido está siempre en funcionamiento, aunque eventualmente puede devolvernos un error, o un valor desactualizado.
  - **(S)** Estado débil (*soft state*): No es necesario que todas los nodos réplica guarden el mismo valor de un ítem en un determinado instante. No existe entonces un “*estado actual de la base de datos*”.
  - **(E)** Consistencia eventual (*eventual consistency*): Si dejaran de producirse actualizaciones, eventualmente todos los nodos réplica alcanzarían el mismo estado.

## ▼ XI - Resolución de Consultas

### Esquema de procesamiento de consultas



1. Parser y Translator
  - Rechazar consultas inválidas
2. Optimizador
  - Conversión a expresión de A.R.
  - Expresión equivalente de A.R.
  - Estrategia para cada operador
3. Evaluación del plan de ejecución
  - Devolver el resultado en base a los datos

### Resolución de cada operador algebraico

- Una vez elegido el árbol de consulta a utilizar, se debe elegir de qué modo resolver cada operador algebraico
- Existen distintos algoritmos para cada operador cuyo costo depende de muchos factores:
  - Cantidad de datos
  - Tamaño de datos
  - Existencia de índices
  - Memoria disponible

### Estimación del costo

- Para poder elegir un método sobre otro, el SGBD estima cuánto le costará resolver la consulta con cada método y elige el de menor costo estimado.
  - La estimación tiene un costo mucho menor a ejecutar la consulta
- Para esto, tiene información sobre los datos de las tablas
  - Guarda estos metadatos en el catálogo
  - PostgreSQL usa la tabla pg\_statistics y su vista pg\_stats

## Catálogo - Información disponible

- Utilizaremos la siguiente notación para la información del catálogo que se posee sobre cada tabla:
  - $n(R)$ : cantidad de filas de la tabla  $R$
  - $B(R)$ : cantidad de bloques que ocupa la tabla  $R$
  - $F(R)$ : (factor de bloque) cantidad de filas por bloque en la tabla  $R$ . Se puede calcular como  $n(R) / B(R)$
  - $V(A, R)$ : (variabilidad de  $A$  en  $R$ ) cantidad de distintos valores que tiene el atributo  $A$  en la tabla  $R$
- También información sobre índices
  - $Height(I(A, R))$ : Altura del índice, para índices de tipo árbol

Mantener el catálogo actualizado no es sencillo, los motores suelen hacerlo con cierta periodicidad, o cuando están ociosos, a veces incluso actualizan la información sobre un muestreo de los datos y no sobre todos ellos. Igualmente valores aproximados son útiles para hacer buenas estimaciones de costos.

## Índices

En un libro me dicen en qué página está cada tema. En un SGBD en qué bloques están las filas con un cierto valor en una más columnas.

Se puede indexar los datos de una tabla por una o más expresiones. El índice, generalmente implementado como un árbol B+, guardará para cada posible valor de las expresiones, en qué bloque o bloques hay filas con ese valor. Esto agiliza la búsqueda por esas expresiones y también por rangos. En índices por varias expresiones, sólo agiliza en búsquedas que las usan en el orden definido.

Tener índices empeora la performance de los cambios en la base.

## Árboles B+

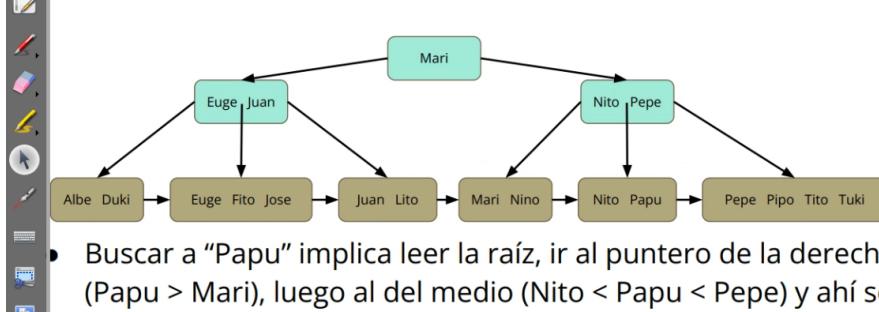
Los árboles B+ tienen ciertas reglas de actualización que les dan prioridades buenas para ser usados como índices.

- Salvo el nodo raíz, todos los nodos se encuentran con al menos un 50% de ocupación
- Todos los nodos hoja tienen la misma altura
- Operaciones de ABM intentan reorganizar lo menos posible los nodos del árbol

## Tipos de nodo

Tienen 2 tipos de nodo, nodos hoja e internos, la cantidad de claves que entran por nodo dependen del tamaño del nodo y del tamaño de las claves a agregar al árbol. Cada nodo interno tiene  $k$  claves y  $k + 1$  punteros, los cuales apuntan a nodos de un nivel inferior. Cada nodo hoja tiene  $k$  claves y  $k + 1$  punteros,  $k$  punteros uno por cada clave y 1 puntero extra al siguiente nodo hoja en orden.

## Índices - Árboles B+ - Ejemplo de búsqueda



### Costo de búsqueda

1. La altura del índice (este costo es igual para cualquier clave buscada)
2. La cantidad de bloques en el que haya filas con el valor buscado (este costo depende del valor buscado).

### Clustering

- Algunos motores tienen la opción de hacer índices de clustering, en los cuales los datos físicamente están ordenados por la clave del índice
- Esto permite reducir el costo de acceso
  - Si un valor del índice está en 5 filas y entran 10 filas por bloque, esas 5 filas casi seguro estarán en un único bloque si el índice es de clustering
  - Si no es de clustering, probablemente estarán en 5 bloques distintos

### Costos de Operadores

El costo lo mediremos en bloques de disco accedidos, el costo real puede diferir del estimado. Las estimaciones se hacen en base a suposiciones que no coinciden 100% con la realidad. En la mayoría de los casos igualmente el costo estimado permitirá elegir el método que será óptimo al ejecutarse.

#### Selección:

- $B(R)$
- índice primario:  $\text{Height}(I(A, R)) + 1$
- índice secundario:  $\text{Height}(I(A, R)) + \lceil n(R) / V(A, R) \rceil$
- índice de clustering:  $\text{Height}(I(A, R)) + \lceil B(R) / V(A, R) \rceil$ .

#### Histogramas

Para mejorar la estimación, se utilizan histogramas que guardan la frecuencia de los valores que tiene cada columna. A veces no todos, sino los  $N$  más frecuentes, es útil para valores discretos y con repeticiones.

#### Proyección

- sin DISTINCT es  $B(R)$
- con DISTINCT sobre superclave también

- con DISTINCT deja de ser  $B(R)$ ... es  $2 * B(R) * \text{ceil}(\log_{M-1}(B(R))) - B(R)$

Cuando el resultado no entra en memoria y hay que eliminar duplicados, con leer la tabla no alcanza. La última fila puede producir un valor que ya se envió con la primera fila. Para resolver este problema, podemos primero ordenar la tabla por el atributo de proyección con un sort externo, esta operación tiene un costo mayor, en la última etapa, al leerse en forma ordenada los datos, se devuelve el primero de cada uno de ellos.

- Tenemos  $M$  bloques de memoria disponibles
- **Primera etapa:** Generamos particiones ordenadas de  $M-1$  bloques en memoria (sort interno)
- **Segunda etapa:** Recorremos  $M-1$  particiones ordenadas a la vez, y generamos una única partición con los datos ordenados de esas particiones (Merge).

La cantidad total de etapas está dada por  $\log_{M-1}(B(R))$ , siempre se escribe y lee  $B(R)$  salvo en la última etapa que solo se lee. Definimos un costo de ordenamiento basado en  $M$  bloques de memoria:  $2 * B(R) * \text{ceil}(\log_{M-1}(B(R))) - B(R)$

- Para resolver la proyección con sort externo, salvo la primera lectura, no se trabaja con toda la fila
  - Se trabaja con las columnas proyectadas
  - Tienen un  $B(\pi X(R)) < B(R)$
- Adaptamos la fórmula trabajando con el  $B(\pi X(R))$ 
  - Sumamos  $B(R)$  y restamos  $B(\pi X(R))$  para compensar

$$\text{Cost}(B(\pi X(R))) = 2 * B(\pi X(R)) * \lceil \log_{M-1}(B(\pi X(R))) \rceil - 2 * B(\pi X(R)) + B(R)$$

Acá se restó  $B(\pi X(R))$  y se sumó  $B(R)$

## Join

Suele ser la operación más demandante. Existen 4 alternativas:

- Loops anidados por bloque
- Loop con único índice
- Sort-merge
- Junta Hash grace

### Loops anidados por bloque

costo:  $\min(B(R), B(S)) + B(R) * B(S)$ , mejorado es:  $B(R) + \text{ceil}(B(R) / (M-2)) * B(S)$

### Loop con único índice

Cuando la condición es de igualdad y una tabla (S) tiene un índice sobre los campos de la condición, puede llegar a aprovecharse este índice. Se recorre bloque a bloque la otra tabla (R) y para cada fila, se hace un index scan en la tabla con índice (S).

Si se usa el índice I sobre el atributo A de la tabla S y no es de clustering, el costo es:

$$B(R) + n(R) * (\text{Height}(I(A, S)) + \text{ceil}(n(S) / V(A, S)))$$

En cambio, si el índice es de clustering:

$$B(R) + n(R) * (\text{Height}(I(A, S)) + \text{ceil}(B(S) / V(A, S)))$$

Si es primario:

$$B(R) + n(R) * (\text{Height}(I(A, S)) + 1)$$

### Sort-Merge

Cuando la condición es de igualdad y las tablas están ordenadas por los atributos de la condición, se puede ir procesando bloque a bloque de modo similar a los operadores de conjunto. La limitante de memoria sería el poder almacenar todos los bloques que tienen un mismo valor en los atributos de la condición, el costo vendría por leer bloque a bloque ambas tablas.

Como en general las tablas no están ordenadas por los atributos del join, podemos pensar el costo del sort-merge como la suma de:

- Ordenar la tabla R y guardar el resultado
- Ordenar la tabla S y guardar el resultado
- Lee los resultados ordenados y procesar el join

Si alguna estuviera ordenada, se ahorra el costo de ordenar y de grabar a disco.

costo:  $\text{Cost}(\text{OrdM}(R)) + \text{Cost}(\text{OrdM}(S)) + B(R) + B(S)$

costo de ordenar:  $2 * B(R) * \text{ceil}(\log_{M-1}(B(R)))$

### Junta Hash Grace

Vimos antes que si la tabla más chica entra en memoria (con 2 bloques extra) el costo total es  $B(R) + B(S)$ . Cuando la tabla más pequeña no entra en memoria, sería conveniente poder dividir ambas tablas en distintas particiones de tal modo que siempre las particiones de la menor tabla entren en memoria. Se harían N joins con un costo  $B(R_i) + B(S_i)$  de cada partición.

En una primera etapa, se lee cada tabla, generándose N particiones. Cada fila se guarda únicamente en la partición que corresponde. Se leen y escriben tantos bloques como tenga la tabla.

En una segunda etapa, se hace un join por loops anidados de cada partición. Al entrar en memoria, cada partición tiene un costo de  $B(R_i) + B(S_i)$ , que sumados dan un costo de  $B(R) + B(S)$ .

En total se leyó  $B(R)$  y  $B(S)$ , se escribió  $B(R)$  y  $B(S)$  y luego se leyeron entre todas las particiones nuevamente  $B(R)$  y  $B(S)$  bloques

costo:  $3 * (B(R) + B(S))$

Este costo igualmente está muy limitado por la memoria disponible.

Para poder usar este método, necesitamos definir la cantidad de particiones ( $N$ ) a hacer, y esto está limitado por la memoria disponible ( $M$ ). En la primer etapa voy leyendo cada tabla bloque a bloque, y para acumular las filas en las particiones que corresponde, no puedo usar más de los otros  $M-1$  bloques.

Primer límite:  $N \leq M - 1$

Si la función de hash distribuye bien los valores, cada partición tendrá  $B(R)/N$  y  $B(S)/N$  bloques. Las particiones de la menor tabla deberán entrar completamente en memoria dejando 2 bloques extra para poder usar loops anidados de forma eficiente.

Segundo límite:  $\min(\lceil B(R) / N \rceil, \lceil B(S) / N \rceil) \leq M - 2$

Un tercer límite está definido por la variabilidad de los atributos de la junta. Si la cantidad de valores distintos es menor a  $N$ , no voy a poder llenar  $N$  particiones sino  $V(A, R)$  particiones. Algunas particiones quedarán vacías, y las que tengan datos tendrán un tamaño mayor a  $B(R) / N$  y no entrarán en memoria.

Tercer límite:  $\min(\lceil B(R) / V(A, R) \rceil, \lceil B(S) / V(A, S) \rceil) \leq M - 2$

### Resumen de costos

- Loops anidados:  $B(R) + \lceil B(R) / (M - 2) \rceil * B(S)$
- Único loop con índice:  $B(R) + n(R) * (\text{Costo index scan})$
- Sort Merge:  $2 * B(R) * \lceil \log_{M-1}(B(R)) \rceil + B(R) + 2 * B(S) * \lceil \log_{M-1}(B(S)) \rceil + B(S)$
- Junta Hash Grace:  $3 * (B(R) + B(S))$

### Cardinalidad y cantidad de bloques

La cantidad de filas que salen depende de la cantidad de filas de cada tabla y la distribución de los valores de los atributos de junta. Nunca tendremos más de  $n(R) * n(S)$  filas, pero el join también cambia el factor de bloque  $F$ . Una fila del join tiene los campos de ambas tablas, por ende ocupa más espacio, y entran menos por bloque.

Cardinalidad:  $n(R) * n(S) / \max(V(A, R), V(A, S))$

Si la junta es por varios atributos, entonces hay que considerar en cada tabla la multiplicación de variabilidades, salvo que tuviéramos un conocimiento de cómo se vinculan ambos atributos.

Cardinalidad:  $n(R) * n(S) / \max(V(A1, R) * V(A2, R), V(A1, S) * V(A2, S))$

### Cardinalidad con histogramas

Un histograma me puede ayudar a estimar mejor la cardinalidad. Si un valor de atributo aparece en ambos histogramas, las filas de cada tabla de ese valor se juntaran entre sí, multiplicándose. Si un valor de atributo

aparece en solo uno de ellos, estimar la cantidad de filas en la otra tabla y multiplicar. Para el resto de los atributos, utilizar la fórmula anterior.

### Cantidad de bloques

para estimar el factor de bloque de una fila de  $R \text{ JOIN } S$  podemos pensar que el espacio ocupado es igual al espacio ocupado por una fila de  $R$  más el de una fila de  $S$ . Cada fila de  $R$  ocupa  $1/F(R)$  bloques y cada fila de  $S$ ,  $1/F(S)$  bloques. El inverso de los factores de bloque se puede sumar.

$$1/F(R \text{ JOIN } S) = 1/F(R) + 1/F(S)$$

### Operadores de Conjunto

Para los tres casos, se debe trabajar con ambas tablas ordenadas. Si no entran en memoria y hay que ordenarlas, usar el sort externo, al costo de ordenamiento se le suma grabar a disco la tabla ordenada, luego se lean ambas tablas en orden y se procesan las filas.

**costo:**  $\text{Cost}(\text{OrdM}(R)) + \text{Cost}(\text{OrdM}(S)) + 2 * (B(R) + B(S))$

#### Unión

- Se recorren ordenadas las filas  $r$  y  $s$  de  $R$  y  $S$
- Se deben devolver todas las filas
  - Si  $r = s$  se devuelve una sola de ellas y se avanza sobre ambas tablas hasta que cambien
  - Si son distintas, se devuelve la menor de ellas y se avanza en su tabla hasta que cambie
  - Cuando se llega al final de una tabla, se devuelve todo lo que queda en la otra, sin duplicados

#### Intersección

- Se recorren ordenadas las filas  $r$  y  $s$  de  $R$  y  $S$
- Se deben devolver las filas que están en ambas
  - Si  $r = s$  se devuelve una sola de ellas y se avanza sobre ambas tablas hasta que cambien
  - Si son distintas, se avanza la tabla de la que tiene el menor valor hasta que cambie, sin devolver nada
  - Cuando se llega al final de una tabla, se termina el algoritmo

#### Resta

- Se recorren ordenadas las filas r y s de R y S
- Se deben devolver las filas que están sólo en r
  - Si  $r = s$  se avanza sobre ambas tablas hasta que cambien de valor, sin devolver nada
  - Si  $r > s$  se avanza la tabla S hasta que cambie de valor
  - Si  $r < s$  se devuelve r y se avanza R hasta que cambie de valor
  - Si termina R, se finaliza el algoritmo
  - Si termina S, se devuelven todos los r restantes, sin repetidos, y se termina el algoritmo

### **Cardinalidad y bloques**

Estimar la cardinalidad es difícil ya que no se conoce la cardinalidad de la intersección, en la unión, debería restarse de  $n(R) + n(S)$ . En la resta debería restarse de  $n(R)$ . El factor  $F(R)$  se mantiene, con lo que si supiera la cardinalidad se puede calcular la cantidad de bloques.

## **Pipelining**

Muchas consultas requieren de más de un operador de álgebra relacional para ser resueltas. La entrada de un operador es la salida de otro. Usar  $n(\text{operador})$ ,  $B(\text{operador})$  y  $F(\text{operador})$  para los nuevos cálculos. Los resultados intermedios pueden almacenarse temporalmente en archivos (agrega  $2 * B(\text{operador})$  de costo). Otra opción más eficiente es hacer **pipelining**.

En muchos casos el resultado parcial de un operador puede ser procesado por el siguiente operador de la consulta. No es necesario tener todos los bloques de  $O1(R)$  para calcular  $O2(O1(R))$ . Cuando termine de procesarse  $O1$ ,  $O2$  habrá procesado la salida completa de  $O1(R)$ . Al no tener que materializar la salida de  $O1$ , conviene siempre que se pueda hacer pipelining.

### **Selección a la salida de operadores**

Una selección aplicada a la salida de un operador no agrega costo alguno. Es "hacer un if" con la condición de la selección y descartar las filas que no la cumplen. Cada bloque que sale del operador anterior se copia en memoria, descartando las filas que corresponda. Al llenarse este nuevo bloque se pasa al siguiente operador. no hay acceso a disco, por ende no hay costo.

costo: 0

### **Proyección a la salida de operadores**

Si la proyección no precisa evitar duplicados, se puede hacer sin costo extra a la salida de cualquier operador (costo = 0). Simplemente de cada fila completa recibida en el bloque, se queda con la/s columna/s proyectadas. Si precisa evitar duplicados, puede ir acumulando las proyecciones en memoria hasta ocupar el espacio disponible. Cuando se ocupa todo, se envía a disco ordenado. Se hace sort externo, pero se evita la primer lectura de  $B(R)$ .

costo:  $2 * B(PI_{\{X\}}(R)) * \text{ceil}(\log_{\{M - 1\}}(B(PI_{\{X\}}(R)))) - 2 * B(PI_{\{X\}}(R))$

### **Junta a la salida de operadores**

*Al recibir un bloque del operador anterior, se puede hacer la junta para todas sus filas, sin requerir los próximos bloques. El costo es menor porque se ahorra una lectura de  $B(R)$ .*

- Loops anidados:  $\text{ceil}(B(R) / (M - 2)) * B(S)$
- Único loop:  $n(R) * \text{index\_scan}(S)$
- Junta Hash Grace:  $2 * B(R) + 3 * B(S)$
- Sort merge:  $B(S) + 2 * B(R) * \text{ceil}(\log_{\{M - 1\}}(B(R))) + 2 * B(S) * \text{ceil}(\log_{\{M - 1\}}(B(S)))$

## **▼ XII - Ingeniería de Datos**

### **Bases de Datos Transaccionales**

*Las que estuvimos viendo hasta ahora. A la capacidad de procesar transacciones en línea en forma concurrente entre una gran cantidad de usuarios, y de manera escalable, se le conoce como OLTP (On-line transaction processing).*

*Son en general motores e bases de datos relacionales. Suelen modelarse con un alto nivel de normalización para minimizar la redundancia de datos. utilizan índices para acelerar las búsquedas.*

*El volumen de los datos dinámicos se relaciona con el tamaño de la organización, y debe ser previsto para dimensionar la base de datos: a medida que el negocio crece (en ubicaciones o en cantidad de clientes) nuestro sistema de ventas también debe hacerlo. Esta capacidad de un SGBD para procesar el volumen de datos que la actividad de la empresa genera se denomina **capacidad transaccional**.*

*Ante la posibilidad de extraer información de los datos, se comenzaron a desarrollar equipos para la **Inteligencia del Negocio**. Podían realizarse consultas sobre los mismos sistemas que se utilizaban para el flujo del negocio para obtener reportes sobre datos históricos, realizar análisis estadísticos y/o predictivos.*

### **Desglose**

- *ETL Extract, Transform, Load*
- *Data Warehouse*
- *BI (Business Intelligence) / ML (Machine Learning)*

### **ETL Extract, Transform, Load**

*Extraer la información de las fuentes. nos vamos a centrar en el caso en el que la fuente de datos es la base de datos transaccional del negocio.*

*Realizar transformaciones sobre los datos para organizar la información de la forma que nos interese que sea consultada.*

*Almacenamiento de la información para consultas futuras.*

## Data Warehouse

*Es el almacén de estos datos que sirve para obtenerlos de diferentes fuentes, para poder analizarlos y sacar información. Lo utilizan los equipos de analistas y científicos de datos.*

## OLAP

*Según Codd, una herramienta de procesamiento analítico debería brindar una serie de servicios que denominó reglas OLAP:*

- Vista conceptual multidimensional: *Mantener los datos en una matriz en la que cada dimensión representa un atributo.*
- Manipulación intuitiva de los datos: *Poder diseñar la vista conceptual a través de una interfaz amigable.*
- Accesibilidad: *Es habitual que las fuentes de datos con que trabajamos sean muy y heterogéneas (distintos SGBDs, formatos). OLAP debería mediar entre las fuentes y la interfaz de usuario.*
- Extracción batch e interpretativa: *Poder almacenar en una base de datos propia el resultado del procesamiento en batch, y también actualizar ese resultado "en vivo" (real time) si el cliente lo requiere.*
- Modelos de análisis: *Poder responder consultas de tipo estadístico o predictivo que ayuden a la toma de decisiones.*

## Disponibilidad de los datos, Freshness

*Los procesos de extracción de los datos son complejos, y detienen las transacciones cada vez que se ejecutan sobre las bases de datos del negocio. ¿Qué tan seguido se deben ejecutar?. Si se realizaran en horarios no-productivos no detienen al negocio, pero BI no obtiene actualización de datos hasta el día siguiente. Más frecuencia es mejor para BI, implica menos demora en tener información reciente pero es más costoso sobre las bases OLTP del negocio.*

## Modelo Dimensional

*Para construir una estructura OLAP, el SGBD toma una captura instantánea de la base de datos en un momento determinado, y agrega los datos para construir un **modelo multidimensional** de los mismos.*

## Jerarquías de dimensiones

*En general cada dimensión tiene una serie de atributos asociados, que podemos representar a través de una relación. A las tablas que describen las dimensiones en un data warehouse se les denomina **tablas de dimensiones**.*

## Operaciones OLAP

*La operación de **roll-up** consiste en agregar los datos de una dimensión, subiendo un nivel en su jerarquía.*

*La operación contraria a roll-up es el **drill-down**. En este caso bajamos un nivel en la jerarquía de una de las dimensiones.*

*La operación de **pivoteo** consiste en producir una tabla agregada por un subconjunto del conjunto de dimensiones en cierto orden deseado.*

*La operación de **slicing** y **dicing** permiten realizar una selección en una dimensión (feta, o slice) o en más de una dimensión (dado, o dice).*

## Data Lake

### Primera definición

*Es un lugar donde se centralizan todos los datos para ser explorados por quien los necesite. Pueden almacenarse en distintos formatos, lugares y tener distintos dueños en el negocio.*

### Segunda definición

*Siendo cada vez menor el costo de almacenamiento de sistemas de objetos (clave-valor) y archivos distribuidos (HDFS, S3, GCS, ...). Se comenzaron a almacenar datos sin formato en los llamados **Data Lakes**.*