Ignacio Méndez Pérez
2023

Resumen Bases de Datos

INDICE

Ignacio Méndez Pérez
2023

Álgebra Relacional

El objetivo principal del álgebra relacional es proporcionar un conjunto de operaciones que permitan realizar consultas y manipulaciones sobre tablas o relaciones en una base de datos relacional. Estas operaciones se basan en la teoría de conjuntos y álgebra matemática, lo que permite una base sólida y consistente para trabajar con datos relacionales.

- **Proyección** π: Selecciona columnas específicas de una tabla y elimina las demás. Es equivalente al "SELECT".

$$\pi_{año}(Películas)$$

- **Selección** σ: Permite seleccionar las filas que cumplen con una condición específica. Es el equivalente al "WHERE".

$$\pi_{nombre,calificación}\left(\sigma_{calificación<8.5}(Películas)\right)$$

- **Unión** $\cup$: Combina dos tablas para obtener un conjunto de tuplas distintas.

$$\pi_{nombre}(Actores) \cup \pi_{director}(Películas)$$

- **Renombrar atributo** ρ: Podemos cambiar nombres de atributos en una relación, así como también podemos cambiar nombres de relaciones (y usarlas más tarde).

$$\rho\big((nombre \to name, año \to year, categoria \to category), peliculas\big)$$

Modifica los atributos de la tabla de actores

$$\rho\big(actores\_jovenes, \sigma_{edad<30}(actores)\big)$$

Guardamos en actores_jovenes nuestros nuevos datos

Luego, si queremos consultar los nombres de esos actores:

$$\pi_{nombre}(actores\_jovenes)$$

- **Producto cruz** $\times$: Combina cada tupla de una tabla con todas las tuplas de otra tabla.

$$\sigma_{peliculas.id=actuo\_en.id\_pelicula}(peliculas \times actuo\_en)$$

- **Join** ⋈: Es básicamente un producto cruz que selección filas según la condición que se le especifique:

$$R_1 \bowtie_{condicion} R_2 = \sigma_{condicion}(R_1 \times R_2)$$

  Cuando los atributos en ambas relaciones tienen el mismo nombre, es posible **no indicar la condición**:

- **Intersección** ∩: Obtiene las tuplas que están presentes en ambas tablas.

$$\rho\left(R, R_1 \bowtie_{R_1.a_1=R_2.a1 \wedge \cdots \wedge R_1.a_n=R_2.a_n} R_2\right)$$
$$R_1 \cap R_2 = \pi_{R_1.a_1 \wedge \cdots \wedge R_2.a_n}$$

- **Diferencia** −: Obtiene las tuplas presentes en una tabla pero no en la otra.

$$\pi_{nombre}(peliculas\_nolan \; - peliculas\_inarritu)$$

SQL (*Structured Query Language*)

Esquema: Especifica nombre de la tabla y sus atributos.

$$Películas(id, nombre, año, categoría, calificación)$$

Instancia: Una instancia de un esquema es un conjunto de tuplas para cara relación del esquema.

ID Película	Nombre Película	Año	Categoría	Calificación (IMDB)
1	Interstellar	2014	Fantasía	8.6
2	The Revenant	2015	Drama	8.1
3	The Imitation Game	2014	Biografía	8.1
4	The Theory of Everything	2014	Biografía	7.7

Llaves: es una restricción que se impone a un esquema.

- **Super llave (*superkey*)**: Conjunto de atributos de R tal que no pueden existir dos tuplas en R con los mismos valores de estos atributos.
- **Llave (candidata)**: Conjunto de atributos de R que es una super llave de R, y no existe un subconjunto propio de estos atributos que sea una super llave. Es decir, es una super llave que no se puede "achicar más".
- **Llave primaria**: Es una llave candidata que queremos destacar y la subrayamos en el esquema.
- ***Surrogate key***: Clave artificial o creada específicamente para actuar como identificador único en una tabla

RDBMS: *Relational Data Base Management System*

Ejemplos de consultas en SQL básico:

VARCHAR (*Variable Character*): almacena cadenas de caracteres de longitud variable.

```
CREATE TABLE Peliculas(
    id int PRIMARY KEY,
    nombre varchar(30),
    año int,
    categoria varchar(30) DEFAULT 'Acción',
    calificacion float DEFAULT 0,
    director varchar(30)
)
```

```
DROP TABLE Peliculas
```

```
ALTER TABLE Peliculas DROP COLUMN director
```

```
ALTER TABLE Peliculas ADD COLUMN productor varchar(30)
```

```
INSERT INTO Peliculas(id, nombre, año, categoria, calificacion, director) VALUES (321351, 'V for Vendetta', 2005,'Action', 8.2 ,'James McTeigue')
```

```
INSERT INTO Peliculas VALUES (321351, 'V for Vendetta', 2005,'Action', 8.2 ,'James McTeigue')
```

```
SELECT a_1, ..., a_n
FROM T_1, ..., T_m
WHERE <condicion>
```

$$\pi_{a_1,\ldots,a_n}(\sigma_{condiciones}(T_1 \times \cdots \times T_m))$$

```
UPDATE Peliculas
SET calificacion = 0
WHERE name = 'Sharknado 6'
```

```
DELETE FROM Tweets
WHERE user_name ='realDonaldTrump'
```

```
SELECT (nombre || ' dirigida por ' || director) as creditos, año
FROM Peliculas
```


creditos	año
V for Vendetta dirigida por James McTeigue	2005
Dunkirk dirigida por C. Nolan	2017

```
SELECT nombre, calificacion
FROM Peliculas
ORDER BY nombre DESC, calificacion
```

En caso de que haya un empate con nombre, entonces se resuelve con el atributo de calificación.

Operadores de conjuntos:

- EXCEPT: Diferencia del álgebra.
- UNION: Unión del álgebra.
- INTERSECT: Intersección del álgebra.
- UNION ALL: Unión que admite duplicados.

```
SELECT nombre
FROM Actores
UNION
SELECT director
FROM Peliculas
```

```
SELECT *
FROM Peliculas
WHERE name LIKE '%Spiderman%'
```

```
SELECT DISTINCT nombre
FROM Peliculas
```

- '%' indica que puede haber cualquier secuencia de caracteres.
- '–' indica que puede haber cualquier carácter (solo uno).

Entrega todos los nombres distintos de las películas

SQL avanzado:

- **AVG(atributo)**: Se utiliza para calcular el promedio de los valores de la columna especificada.

```
SELECT AVG(precio)
FROM Productos
WHERE fabricante = 'Toyota'
```

Entregaría el promedio de la columna `precio` de la tabla `productos` de las filas que cumplan la condición.

- **COUNT(atributo)**: Cuenta el número de filas donde el valor del atributo especificado **no** es nulo. Se puede llamar a COUNT(*) para contar todas las filas en una tabla, independientemente de si hay valores nulos en alguna columna.

```
SELECT COUNT(*)
FROM Productos
WHERE año > 2012
```

```
SELECT COUNT(fabricante)
FROM Productos
WHERE año > 2012
```

```
SELECT DISTINCT COUNT(cantidad)
FROM Compra
```

Se utiliza incorrectamente. Esto es lo equivalente a realizar la consulta de SELECT COUNT(cantidad) FROM Compra

```
SELECT COUNT(DISTINCT cantidad)
FROM Compra
```

Entrega la cantidad de atributos diferentes que existen en la tabla. Es la forma correcta de combinar las operaciones.

- **SUM(atributo)**: Obtiene la suma de los valores en una columna especificada.
- **MIN(atributo)**: Obtiene el valor mínimo de una columna numérica o alfabética.
- **MAX(atributo)**: Obtiene el valor máximo de una columna numérica o alfabética.
- **GROUP BY**: Se utiliza para agrupar filas que comparten el mismo valor en una o más columnas. Esta cláusula permite realizar operaciones de agregación, como SUM, COUNT, AVG, MIN y MAX, en grupos de filas en lugar de en el conjunto de datos completo. La sintaxis básica de GROUP BY es la siguiente:

```
SELECT columna1, columna2, ... , columnaN, función_agregación(columnaX)
FROM nombre_tabla
GROUP BY columna1, columna2, ... , columnaN;
```

Ejemplo:

```
SELECT fabricante, COUNT(fabricante)
FROM Productos
WHERE año > 2012
GROUP BY fabricante
```

Consulta los resultados según el FROM y WHERE, agrupa los resultados según los atributos de GROUP BY, y para cada grupo se aplica independientemente la agregación.

Da como resultado una tabla en donde la primera columna es el nombre del `fabricante` y la segunda columna es la cantidad de veces que aparece el fabricante en la tabla, es decir, la cantidad de productos que tiene el fabricante.

Otro ejemplo:

```
SELECT producto, SUM(precio*cantidad) as ventaTotal
FROM Compra
WHERE fecha > '10/01'
GROUP BY producto
```

1) Se computa el FROM y el WHERE:

producto	fecha	precio	cantidad
tomates	07/02	100	6
tomates	06/07	150	4
zapallo	08/02	800	1
zapallo	09/07	1000	2

2) Se agrupa según GROUP BY:

producto	fecha	precio	cantidad
tomates	07/02	100	6
	06/07	150	4
zapallo	08/02	800	1
	09/07	1000	2

3) Se agrega por grupo y se ejecuta la proyección:

producto	ventaTotal
tomates	1200
zapallo	2800

- **HAVING**: Misma consulta, pero sólo queremos los productos que se vendieron más de 100 veces.

```
SELECT producto, SUM(precio*cantidad) AS ventaTotal
FROM Compra
WHERE fecha > '10/01'
GROUP BY producto
HAVING SUM(cantidad) > 100
```

¿Por qué usamos HAVING y no lo incluimos en el WHERE?

R: HAVING se aplica después de la agrupación y las funciones de agregación, mientras que WHERE se aplica antes de la agrupación.

- **Consultas anidadas**

  - **IN**: Se utiliza para comprobar si un valor coincide con cualquiera de los valores de una lista o subconsulta.
    En el ejemplo comprobamos que `Bandas.nombre` esté dentro del listado de bandas que han tocado en Lollapalooza.

```
SELECT Bandas.nombre
FROM Bandas, Estudiantes_UC
WHERE Bandas.vocalista = Estudiantes_UC.nombre
AND Bandas.nombre IN (
    SELECT Toco_en.nombre_banda
    FROM Toco_en
    WHERE Toco_en.nombre_festival = 'Lollapalooza'
)
```

  - **ALL**: Se utiliza para comparar un valor con todos los valores de una lista o subconsulta y devuelve `true` si se cumple para todos ellos.
    En el ejemplo obtenemos las cervezas cuyo precio sea **menor** a las "Austral Calafate".

```
SELECT Cervezas.nombre
FROM Cervezas
WHERE Cervezas.precio < ALL (
    SELECT Cervezas2.precio
    FROM Cervezas AS Cervezas2
    WHERE Cervezas2.nombre = 'Austral Calafate'
)
```

  - **ANY**: Se utiliza para comparar un valor con al menos uno de los valores de una lista o subconsulta y devuelve `true` si se cumple para al menos uno de ellos.
    En el ejemplo obtenemos las cervezas cuyo precio NO sea la más cara.

```
SELECT Cervezas.nombre
FROM Cervezas
WHERE Cervezas.precio < ANY (
    SELECT Cervezas2.precio
    FROM Cervezas AS Cervezas2
)
```

    Podemos usar el operador <> para denotar el "distinto".
    En el ejemplo podemos obtener los nombres de películas que repiten en años diferentes.

```
SELECT p.nombre
FROM Películas AS p
WHERE p.año <> ANY (
    SELECT año
    FROM Películas
    WHERE nombre = p.nombre
)
```

  - **EXISTS**: Se utiliza para comprobar si una subconsulta devuelve algún resultado.
    En el ejemplo obtenemos los nombres de los clientes que han realizado al menos un pedido.

```
SELECT nombre
FROM Clientes c
WHERE EXISTS (
    SELECT 1
    FROM Pedidos p
    WHERE p.id_cliente = c.id_cliente
);
```

- ***Inner Joins***: Estas 3 consultas son equivalentes:

```
SELECT *
FROM Peliculas,
Actuo_en
WHERE id = id_pelicula
```

```
SELECT *
FROM Peliculas JOIN Actuo_en
ON id = id_pelicula
```

```
SELECT *
FROM Peliculas INNER JOIN Actuo_en
ON id = id_pelicula
```

- ***Outer Joins***:

  - **`LEFT OUTER JOIN`**: Devuelve todas las filas de la tabla de la izquierda y las filas coincidentes de la tabla de la derecha. Si no hay coincidencias en la tabla de la derecha, se muestran valores nulos para las columnas de la tabla de la derecha.

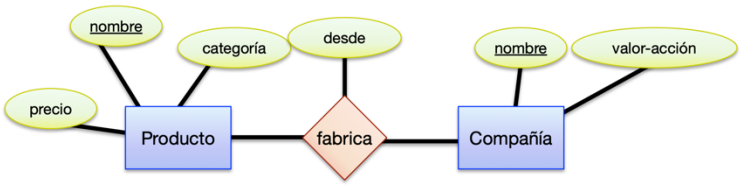
  - **`RIGHT OUTER JOIN`**: Devuelve todas las filas de la tabla de la derecha y las filas coincidentes de la tabla de la izquierda. Si no hay coincidencias en la tabla de la izquierda, se muestran valores nulos para las columnas de la tabla de la izquierda.

  - **`FULL OUTER JOIN`**: Muestra todas las filas de ambas tablas, y cuando no hay coincidencias en una tabla, muestra valores nulos para las columnas de esa tabla.

```sql
SELECT columna1, columna2, ...
FROM tabla1
LEFT OUTER JOIN tabla2 ON tabla1.columnaX = tabla2.columnaY;
```

```sql
SELECT columna1, columna2, ...
FROM tabla1
RIGHT OUTER JOIN tabla2 ON tabla1.columnaX = tabla2.columnaY;
```

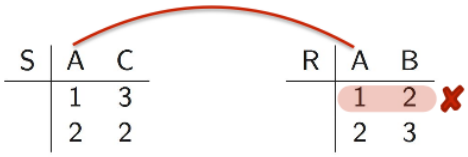
Modelos entidad relación




```sql
CREATE TABLE fabrica(
    p_nombre varchar(30),
    c_nombre varchar(30),
    desde date,
    PRIMARY KEY (p_nombre, c_nombre),
    FOREIGN KEY(p_nombre) REFERENCES producto(nombre),
    FOREIGN KEY(c_nombre) REFERENCES compania(nombre)
)
```

Llaves foráneas

¿Qué ocurre al eliminar (1, 2) en $\boldsymbol{R}$?
3 opciones:



S	A	C
	1	3
	2	2

R	A	B
	1	2
	2	3

1. No permitir eliminación → Obtenemos error.

2. Propagar la eliminación y también borrar (1, 3) en $\boldsymbol{S}$.

```sql
CREATE TABLE S(a int, c int, FOREIGN KEY(a) REFERENCES R ON DELETE CASCADE, ... )
```

3. Mantener la tupa en $\boldsymbol{S}$ pero dejar en la llave foránea el valor `null`.

```sql
CREATE TABLE S(a int, c int, FOREIGN KEY(a) REFERENCES R ON DELETE SET NULL, ... )
```

Restringir el dominio de columnas:

En este caso verificamos que el `precio` esté entre `10.000` y `1.000.000`, y que la `fecha_fin` sea más tarde que la `fecha_inicio`.

```
CREATE TABLE Festival(
    id int PRIMARY KEY,
    nombre varchar(30) NOT NULL,
    fecha_inicio date NOT NULL,
    fecha_fin date NOT NULL,
    precio int NOT NULL,
    CHECK( precio BETWEEN 10000 AND 1000000 ),
    CHECK( fecha_fin > fecha_inicio)
)
```

***First Normal Form* (1NF)**

- Una celda no debe tener más de 1 valor (atomicidad).
- Debe haber una llave primaria.
- No debe haber columnas ni filas duplicadas.
- Cada columna debe tener 1 solo valor para cada fila.

EMPLOYEE_ID	NAME	JOB_CODE	JOB	STATE_CODE	HOME_STATE
E001	Alice	J01	Chef	26	Michigan
E001	Alice	J02	Waiter	26	Michigan
E002	Bob	J02	Waiter	56	Wyoming
E002	Bob	J03	Bartender	56	Wyoming
E003	Alice	J01	Chef	56	Wyoming

***Second Normal Form* (2NF)**

- Está en 1ND.
- No tiene dependencias parciales. Esto significa que cada atributo de la tabla que no sea una llave depende completamente de la llave primaria.

EMPLOYEE_ID	JOB_CODE
E001	J01
E001	J02
E002	J02
E002	J03
E003	J01

EMPLOYEE_ID	NAME	STATE_CODE	HOME_STATE
E001	Alice	26	Michigan
E002	Bob	56	Wyoming
E003	Alice	56	Wyoming

JOB_CODE	JOB
J01	Chef
J02	Waiter
J03	Bartender

***Third Normal Form* (3NF)**

- Está en 2NF
- No tiene dependencias transitivas. Esto significa que no debe haber dependencias funcionales no triviales entre los atributos que no son llaves.
- Una relación $\boldsymbol{R}$ está en $\boldsymbol{3NF}$ si para toda dependencia funcional no trivial $X \rightarrow Y$, $\boldsymbol{X}$ es una superllave o $\boldsymbol{Y}$ es parte de una llave minimal.

EMPLOYEE_ID	JOB_CODE
E001	J01
E001	J02
E002	J02
E002	J03
E003	J01

EMPLOYEE_ID	NAME	STATE_CODE
E001	Alice	26
E002	Bob	56
E003	Alice	56

JOB_CODE	JOB
J01	Chef
J02	Waiter
J03	Bartender

STATE_CODE	HOME_STATE
26	Michigan
56	Wyoming

***Boyce-Codd Normal Form* (BCNF)**

- Una tabla está en BCNF si, para cada dependencia funcional no trivial ($A \rightarrow B$) en la tabla, $\boldsymbol{A}$ es una super llave. Esto significa que todas las dependencias funcionales no triviales deben tener una llave completa como su lado izquierdo.
- BCNF garantiza que cada dependencia funcional esté basada en una clave completa, lo que evita cualquier posible redundancia funcional.

Cursores en Python

```
cur = db_connection.cursor() # instanciamos el cursor
create_query = "CREATE TABLE Peliculas(id INT, titulo
VARCHAR(100));"
insert_query = "INSERT INTO Peliculas VALUES (1, 'Minions');"
try:
   cur.execute(create_query)
   cur.execute(insert_query)
except sqlite3.OperationalError as e:
   print(e)

db_connection.commit() # guardamos los cambios en disco.
db_connection.close() #cerramos la conexion
```

¿Por qué no concatenamos strings para generar consultas?

```
query = "SELECT * FROM usuarios WHERE nombre = " + nombre
```

R: Nos exponemos a SQL *injection*.
Alternativa:

```
query = "SELECT * FROM usuarios WHERE nombre = %(nombre)s"

cursor.execute(query, {'nombre': nombre})
```

ORM (*Object Relational Mapping*)

Técnica de programación que permite relacionar de forma automática y transparente objetos en un lenguaje de programación orientado a objetos con las tablas de una base de datos relacional.

Es una abstracción que facilita el manejo y la interacción con la base de datos, permitiendo que los desarrolladores trabajen principalmente con objetos y clases en su código, en lugar de escribir consultas SQL directamente.

Algoritmos

- **Selección σ_p**: Lee todas las tuplas una a una, sin índices. Retorna la tupla si satisface el predicado (p).

Algoritmo

input: Predicado p y relación (e.g. operador) R.

```
open()
    R.open()

close()
    R.close()

next()
    t := R.next()
    while t ≠ NULL do
        if p(t) = true then
            return t
        t := R.next()
    return NULL
```

- **Proyección π_L**: Lee las tuplas una a una. Proyectamos en los atributos L.

Algoritmo

input: Lista de atributos L y operador R.

```
open()
    R.open()

close()
    R.close()

next()
    t := R.next()
    if t ≠ NULL then
        return t.project(L)
    return NULL
```

- ***Nested Loop Join***: Por cada tupla de la primera relación, se leen todas las tuplas de la segunda relación

Algoritmo

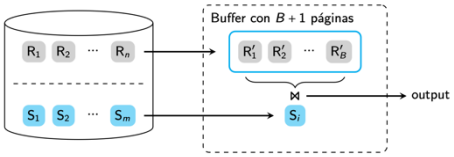
input: Operadores R y S, y un predicado p.

```
open()
    R.open()
    S.open()
    r := R.next()

close()
    R.close()
    S.close()

next()
    while r ≠ NULL do
        s := S.next()
        if s = NULL then
            S.close()
            r := R.next()
            S.open()
        else if (r, s) satisfacen p then
            return (r, s)
    return NULL
```

- ***Block Nested Loop Join***: En lugar de procesar un solo registro de A y un solo registro de B en cada iteración, se procesa un bloque (o grupo) de registros de la tabla externa A junto con un bloque correspondiente de registros de la tabla interna B.



Algoritmo

input: Operadores R y S, un predicado p, y un **Buffer** con $B+1$ páginas.

```
open()
    R.open()
    fillBuffer()

close()
    R.close()
    S.close()

fillBuffer()
    Buff := ∅
    r := R.next()
    while r ≠ NULL do
        Buff := Buff ∪ {r}
        if Buff.isFull() then
            break
        r := R.next()
    S.open()
    s := S.next()
```

Algoritmo

input: Operadores R y S, un predicado p, y un **Buffer** con $B+1$ páginas.

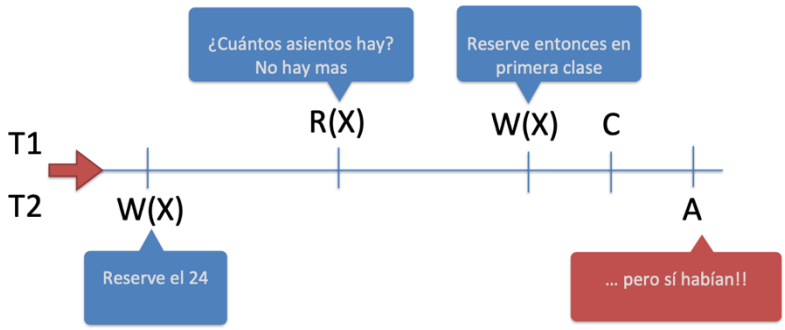
```
next()
    while Buff ≠ ∅ do
        while s ≠ NULL do
            r := Buffer.next()
            if r = NULL then
                Buff.reset()
                s := S.next()
            else if (r, s) ⊨ p then
                return (r, s)
        fillBuffer()
    return NULL
```

Transacciones ACID

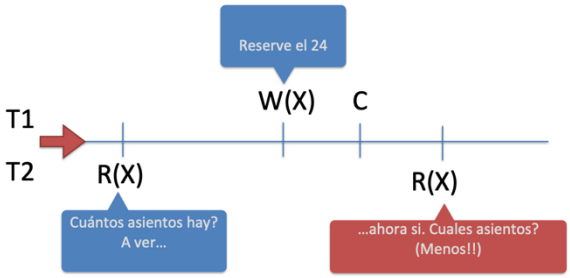
- ***Atomicity***: O se ejecutan todas las operaciones de la transacción, o no se ejecuta ninguna.
- ***Consistency***: Garantiza que una transacción llevará la base de datos desde un estado válido a otro estado también válido.
- ***Isolation***: Asegura que cada transacción se ejecute de manera independiente y no se vea afectada por otras transacciones que se estén ejecutando simultáneamente.
- ***Durability***: Los cambios que hace cada transacción son permanentes en el tiempo, independiente de cualquier tipo de falla.

Conflictos en transacciones

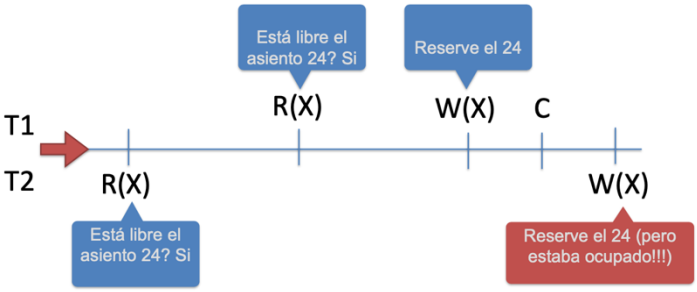
- Lecturas sucias (*Write – Read*)



- Lecturas no repetibles (*Read – Write*)



- Actualización perdida o reescritura de datos temporales (*Write – Write*).



Schedule

Un ***schedule*** S es una secuencia de operaciones primitivas de una o más transacciones, tal que, para toda transacción T_i , las acciones de T_i aparecen en S en el mismo orden de su definición.

Un *schedule* es **serial** si todas las acciones de cada transacción son ejecutas en grupo y no hay intercalación de acciones.

Schedule serial

T_1	T_2	A	B
READ(A,x)		100	50
$x := x + 100$			
WRITE(A,x)		200	
READ(B,x)			
$x := x + 200$			
WRITE(B,x)			250
	READ(A,y)		
	$y := y * 2$		
	WRITE(A,y)	400	
	READ(B,y)		
	$y := y * 3$		
	WRITE(B,y)	400	750

Schedule NO serial

T_1	T_2	A	B
READ(A,x)		100	50
$x := x + 100$			
WRITE(A,x)		200	
	READ(A,y)		
	$y := y * 2$		
	WRITE(A,y)	400	
READ(B,x)			
$x := x + 200$			
WRITE(B,x)			250
	READ(B,y)		
	$y := y * 3$		
	WRITE(B,y)	400	750

Un schedule S es **serializable** si existe algún serial schedule S' tal que el resultado de S y S' es el mismo para todo estado inicial de la BD.

Schedule **serializable**. Da lo mismo con los datos que empieza

T_1	T_2	A	B
READ(A,x)		100	50
$x := x + 100$			
WRITE(A,x)		200	
	READ(A,y)		
	$y := y * 2$		
	WRITE(A,y)	400	
READ(B,x)			
$x := x + 200$			
WRITE(B,x)			250
	READ(B,y)		
	$y := y * 3$		
	WRITE(B,y)	400	750

Schedule **NO serializable**. T_2 está leyendo el valor de A, y se modifica B antes que T_1

T_1	T_2	A	B
READ(A,x)		100	50
$x := x + 100$			
WRITE(A,x)		200	
	READ(A,y)		
	$y := y * 2$		
	WRITE(A,y)	400	
	READ(B,y)		
	$y := y * 3$		
	WRITE(B,y)		150
READ(B,x)			
$x := x + 200$			
WRITE(B,x)		400	350

La tarea del ***Transaction Manager*** es permitir solo *schedules* que sean **serializables**.

Acciones conflictivas (para 2 transacciones i, j):

- $P_i(X), Q_i(Y)$ con $P, Q \in \{R, W\}$.
- $R_i(X), W_j(X)$
- $W_i(X), R_j(X)$
- $W_i(X), W_j(X)$

Acciones NO conflictivas (para 2 transacciones i, j):

- $R_i(X), R_j(Y)$.
- $R_i(X), W_j(Y)$ con $X \neq Y$.
- $W_i(X), R_j(Y)$ con $X \neq Y$.
- $W_i(X), W_j(Y)$ con $X \neq Y$.

Podemos cambiarlas de orden en un **schedule**!

T_1	T_2
$R(A)$	
	$R(A)$
	$R(C)$
$R(B)$	
$W(A)$	
	$W(C)$
$R(D)$	

T_1	T_2
	$R(A)$
$R(A)$	
	$R(C)$
$R(B)$	
$W(A)$	
	$W(C)$
$R(D)$	

T_1	T_2
	$R(A)$
$R(A)$	
$R(B)$	
	$R(C)$
$W(A)$	
	$W(C)$
$R(D)$	

T_1	T_2
	$R(A)$
$R(A)$	
$R(B)$	
$W(A)$	
	$R(C)$
	$W(C)$
$R(D)$	

T_1	T_2
	$R(A)$
$R(A)$	
$R(B)$	
$W(A)$	
	$R(C)$
$R(D)$	
	$W(C)$

Puedo permutar un par de operaciones consecutivas si:

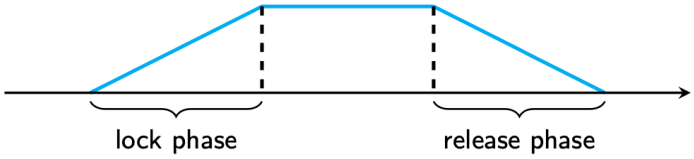
- No usan el mismo recurso.
- Usan el mismo recurso pero ambas son de lectura.

Un *schedule* es **conflict serializable** si puedo transformarlo a uno **serial** usando permutaciones.

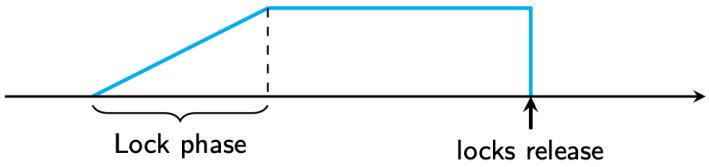
Si un schedule es **conflict serializable** implica que también es **serializable**, pero hay schedules serializables que **no** son conflict serializable

2PL (2 *Phase Locking*)

1. ***Lock Phase***: *Locks* son pedidos a medida que son necesitados.
2. ***Release Phase***: *Locks* son liberados si ya no son requeridos.



Strict 2PL



NoSQL

Término que se utiliza para describir un conjunto de bases de datos que difieren en su enfoque y estructura en comparación con las bases de datos relacionales tradicionales. A diferencia de las bases de datos SQL, que se basan en el modelo relacional y utilizan tablas con filas y columnas, las bases de datos NoSQL adoptan modelos de datos no tabulares y más flexibles.

Existen varias razones por las que se utilizan las bases de datos NoSQL:

- **Escalabilidad**: Las bases de datos NoSQL están diseñadas para manejar grandes cantidades de datos y cargas de trabajo distribuidas. Permiten una fácil escalabilidad horizontal agregando más servidores o nodos para gestionar el crecimiento del sistema.

- **Flexibilidad del esquema**: A diferencia de las bases de datos SQL, que requieren un esquema fijo y definido previamente, las bases de datos NoSQL permiten esquemas dinámicos o incluso sin esquema. Esto significa que pueden manejar tipos de datos variables y no se requiere una estructura rígida para cada registro.

- **Rendimiento**: Las bases de datos NoSQL están optimizadas para consultas rápidas y eficientes, especialmente cuando se trata de grandes volúmenes de datos. Eliminan la complejidad de las operaciones JOIN utilizadas en las bases de datos relacionales.

- **Datos no estructurados o semiestructurados**: Si se está trabajando con datos no estructurados, como documentos JSON, XML o datos clave-valor, las bases de datos NoSQL ofrecen una mejor solución, ya que no requieren un esquema estricto.

- **Alta disponibilidad y tolerancia a fallos**: Muchas bases de datos NoSQL están diseñadas para ser altamente disponibles y tolerantes a fallos. Esto significa que pueden seguir funcionando incluso si algunos nodos del sistema fallan.