

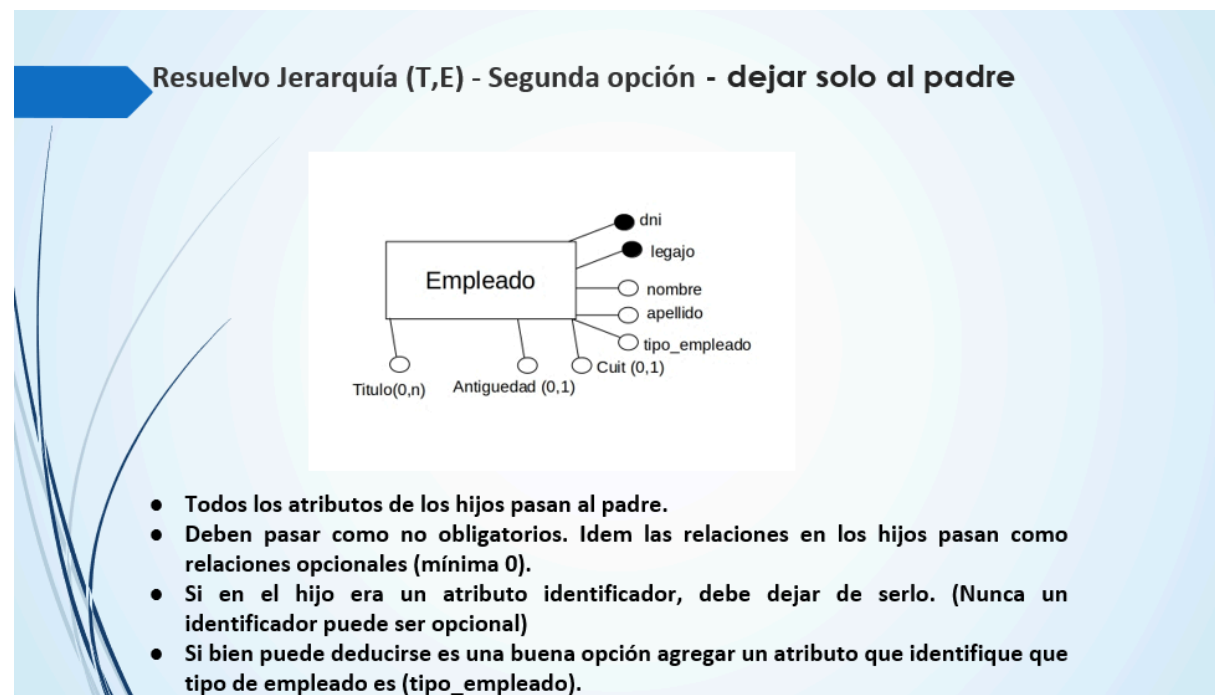
### Modelo conceptual:

La cardinalidad define el grado de relación existente en una agregación.

Un identificador es un atributo o un conjunto de atributos que permite reconocer o distinguir a una entidad de manera unívoca dentro del conjunto de entidades. El concepto de identificador está ligado a los conceptos de claves primarias y candidatas, vertidos en capítulos previos de este libro.

### Modelo logico:

NUNCA UN IDENTIFICADOR PUEDE SER OPCIONAL.



### Modelo fisico.

En casos recursivos creas 1 sola tabla para la entidad.

### Algebra relacional:

Es un lenguaje de manipulación de datos. El Álgebra Relacional (AR) representa un conjunto de operadores que toman las tablas (relaciones) como operandos, y regresan otra tabla como resultado. Basicamente consta de aplicar operaciones entre tablas y obtener una nueva.

El AR puede ser definido formalmente de la siguiente manera. Una expresión básica en AR consta de:

- Una relación o tabla de una BD.
- Una relación o tabla constante (ejemplo de la inserción, un conjunto de tuplas expresada literalmente cada una de ellas).

Una expresión (consulta) general E en AR se construye a partir de subexpresiones (subconsultas) E1, E2, ..., En.

### Optimización de consultas: (optimizador de consultas = DBMS)

el proceso de optimización de consultas comienza con la consulta generada por el usuario, aplicando la siguiente secuencia de pasos:

1. Un analizador sintáctico (parser) genera una expresión manipulable por el optimizador de consultas. El análisis sintáctico convierte al texto de entrada –en este caso, la consulta del usuario– en una estructura tipo árbol, que resulta más útil para su análisis.
2. A partir de la expresión interna, el optimizador obtiene una consulta equivalente más eficiente. En el apartado sobre evaluación del costo de las expresiones, se presenta la comparación entre operaciones equivalentes.
3. Por último, el proceso de optimización considera el estado actual de la BD y los índices definidos, para resolver la consulta que se tiene hasta el momento, con el acceso a disco posible. **Segun lo que dijo el profe, acá se eligen los indices para tener la menor cantidad de accesos a disco posibles y luego de la consulta primera q yo tenia se realiza la consulta alternativa elegida por el DBMS**

TABLA 16.1

Consulta original	Expresión equivalente más eficiente
$\sigma_{\text{predicado}}(\text{tabla1} \times \text{tabla2})$	$(\sigma_{\text{predicado}}(\text{tabla1}) \times \sigma_{\text{predicado}}(\text{tabla2}))$
$\sigma_{\text{predicado 1 AND predicado 2}}(\text{tabla1})$	$\sigma_{\text{predicado 1}}(\sigma_{\text{predicado 2}}(\text{tabla1}))$
$\sigma_{\text{predicado}}(\text{tabla1} \cup \text{tabla2})$	$(\sigma_{\text{predicado}}(\text{tabla1}) \cup \sigma_{\text{predicado}}(\text{tabla2}))$
$\sigma_{\text{predicado}}(\text{tabla1} - \text{tabla2})$	$(\sigma_{\text{predicado}}(\text{tabla1}) - \sigma_{\text{predicado}}(\text{tabla2}))$
$(\text{tabla1} \bowtie \text{tabla2} \bowtie \text{tabla3})$	$((\text{tabla1} \bowtie \text{tabla2}) \bowtie \text{tabla3})$
$(\text{tabla1} \bowtie \text{tabla2})$	$(\text{tabla2} \bowtie \text{tabla1})$

#### Algunos valores:

- CT tabla ,
- CB tabla
- CV (a, tabla)

#### Costo selección: $\sigma_{(at = \text{"valor"})}(\text{Tabla})$

- $(CT \text{ tabla} / CV (at, \text{tabla})) * CB \text{ tabla}$

#### Costo proyección: $\pi_{at1, at2, \dots, atn}(\text{Tabla})$

- $(CB \text{ at1} + CB \text{ at2} + \dots + CB \text{ atn}) * CT \text{ tabla}$

#### Costo producto cartesiano: $T1 \times T2$

- $(CT \text{ t1} * CT \text{ t2}) * (CB \text{ t1} + CB \text{ t2})$

DBD - CLASE 7

**Lo multiplica por el CB (en el primero y ultimo) para calcular el tamaño q va a ocupar eso, pero en realidad esa parte no seria el “costo”.**

pagina 370 y 371, queda demostrado, a partir del análisis realizado, que el producto natural no es conmutativo en cuanto a performance.

Para poder resolver situaciones como la última planteada, el optimizador dispone de estadísticas de la BD. El SGBD mantiene estadísticas con respecto a la cantidad de tuplas de las tablas, así como la distribución de valores para los atributos. De esta forma puede

decidir rápidamente el costo de cada alternativa de solución y aplicar la más conveniente. Las estadísticas se mantienen parcialmente actualizadas. El costo de tenerlas todo el tiempo actualizadas es alto; cualquier inserción, borrado o modificación altera los valores. Para poder efectuar decisiones, basta con tener los datos parcialmente actualizados. Cada cierto tiempo, el gestor de BD ejecuta un proceso de actualización de las estadísticas.

### Seguridad e integridad de datos, teorico 8.1: (audios)

#### Métodos de recuperación de integridad de una base de datos (la perdiste cuando hay un fallo)

El gestor del SGBD es el responsable de controlar la ejecución de las transacciones, de administrar el archivo de bitácora y de utilizar los algoritmos de recuperación cuando el SGBD se recupera de una situación de fallo.

## Registro Historico

<ol style="list-style-type: none"> <li>1. READ ( A, a1)</li> <li>2. a1 := a1 - 100;</li> <li>3. READ (B, b1)</li> <li>4. b1 := b1 + 100;</li> <li>5. WRITE(A, a1)</li> <li>6. WRITE(B, b1)</li> </ol>	<table border="0"> <tr> <th>Bitacora</th> <th>Memoria RAM</th> </tr> <tr> <td>1. &lt;T start&gt;    2 no produce efecto</td> <td>1. A = 1000</td> </tr> <tr> <td>5. &lt;T, A, 900 &gt;    3, 4 sin efecto</td> <td>2. A = 900</td> </tr> <tr> <td>6. &lt;T, B, 2100&gt;</td> <td>3. B = 2000</td> </tr> <tr> <td>7. &lt;T commit&gt;</td> <td>4. B = 2100</td> </tr> </table>	Bitacora	Memoria RAM	1. <T start>    2 no produce efecto	1. A = 1000	5. <T, A, 900 >    3, 4 sin efecto	2. A = 900	6. <T, B, 2100>	3. B = 2000	7. <T commit>	4. B = 2100	<p><b>Base de Datos</b></p> <p>A = 1000 900 (8) B = 2000 2100 (9)</p> <p><b>Base de Datos</b></p> <p>A = 1000 900 (8) B = 2000</p> <p><b>Base de Datos</b></p> <p>A = 1000 B = 2000</p>
Bitacora	Memoria RAM											
1. <T start>    2 no produce efecto	1. A = 1000											
5. <T, A, 900 >    3, 4 sin efecto	2. A = 900											
6. <T, B, 2100>	3. B = 2000											
7. <T commit>	4. B = 2100											

**Plan1** 1, 2, 3, 4, 5, 6, 7, 8, 9 fallo  
Recupera del Fallo y no habria que hacer nada???

**Plan2** 1,2,3,4,5,6,7, 8 fallo  
Recupera del Fallo y que hacemos???

**Plan3** 1,2,3, 4, 5, 6, 7 fallo  
Recupera del fallo y no habria que hacer nada???

**Planotro** fallo antes de 7  
Base de datos bien.

Siempre que ocurra un fallo en una transacción, donde si te fijas en el archivo de la bitácora ese mismo tiene el "T commit" esto implica que la base de datos vos la vas a dejar en condiciones pq se ejecutaron todas las instrucciones de la transacción correctamente por lo que **DEBES** (si hay un fallo como se muestra en esos ejemplos) reejecutar la transición y se te va a hacer todo correctamente más lo que no se habia podido hacer por el fallo. En el ultimo ejemplo que muestra, vos la base de datos la tenes bien pq no se llegó a cargar nada en la base de datos por el fallo antes del paso 7, osea q no se llegó al T commit, por ende vos no guardas nada en la base de datos.

- < To Commit >

Ejecución con To parcialmente cometida, entonces se actualiza la BD.

- No se necesita valor viejo, se modifica la BD al final de la transacción o no se modifica.

Ante un fallo, y luego de recuperarse:

- REDO (Ti), para todo Ti que tenga un Start y un Commit en la Bitácora.
- Si no tiene Commit entonces se ignora, dado que no llegó a hacer algo en la BD.

**ALGO A ACLARAR, LAS ACCIONES DEL ARCHIVO DE BITACORA ES "<Tx, valorA, valor nuevo> .**

Para el caso de modificacion inmediata ahi si tenes que indicar el valor viejo. A su vez, es mas ineficiente con respecto al anterior por el hecho de que hay 2 operaciones posibles frente a fallo, y en el otro hay 1 sola, sin embargo este mismo distribuye mejor la carga de trabajo ya que se va cargando en la base de datos a medida que se ejecutan las instrucciones/operaciones de la transaccion y no lo hace todo de una como el de modificacion diferida. **Como conclusión no hay uno q sea mejor o peor, ambos tienen ventajas y desventajas.**

### Modificación inmediata:

- La actualización de la BD se realiza mientras la transacción está activa y se va ejecutando.
- Se necesita el valor viejo, pues los cambios se fueron efectuando.
- Ante un fallo, y luego de recuperarse:
  - REDO( $T_i$ ), para todo  $T_i$  que tenga un Start y un Commit en la Bitácora.
  - UNDO( $T_i$ ), para todo  $T_i$  que tenga un Start y no un Commit.

13

**Se puede dar una condicion de idempotencia, y esto surge cuando luego de que ocurra un fallo y este se solucione, se debe hacer un REDO o un UNDO, y que cuando lo estas haciendo vuelva a ver otro fallo, en estos casos se sigue asiendo lo mismo, se va a ejecutar un REDO o un UNDO depende el caso no importa cuantos fallos hayan en ese momento.**

El proceso de los buffers es el siguiente, primero se va escribiendo en el buffer de bitácora, dsp en el buffer de la base de datos, se baja el buffer de bitácora a disco y dsp el buffer de la base de datos.

## Registro Historico

### Puntos de verificación:

- Ante un fallo, que hacer
  - REDO, UNDO: según el caso
- Revisar la bitácora:
  - Desde el comienzo?: probablemente gran porcentaje esté correcto y terminado.
  - Lleva mucho tiempo.
- Checkpoints (monousario)
  - Se agregan periódicamente indicando desde allí hacia atrás todo OK.
  - Periodicidad?

15

En este caso, el checkpoint se va a agregar en la bitácora luego de que se almacene lo del buffer de la base de datos a la base de datos /disco) (pq ahí se almacenó ciertas modificaciones hechas por las transacciones q se terminaron de ejecutar en ese momento, pq se llenó el buffer de la bitacora, se pasó al buffer de la base de datos, luego se carga en disco el de la bitacora y cuando se carga el de disco ahí se hace todo esto), entonces se carga el checkpoint que indica que a partir de este punto para atrás no hace falta revisar en caso de que ocurra un fallo ya que todo lo modificado por dichas transacciones ya se cargó correctamente en la base de datos. Dsp del checkpoint para adelante hayq ue aplicar operacion de REDO o UNDO.

### Clase 8.2:

. En un entorno concurrente puede generar situaciones de pérdida de integridad de los datos, aun sin producirse fallos en el procesamiento de las transacciones. Dos transacciones que se ejecutan al mismo tiempo SIN FALLOS en ninguna de ellas pueden dejar la base de datos inconsistente.

**Seriabilidad:** Implica ejecutar una transacción y dsp otra y así sucesivamente y se prueba que garantiza la consistencia de la BD . Abajo tenes 2 planificaciones diferentes, de T0 a T1 y de T1 a T0.

**Seriabilidad**

- Garantiza la consistencia de la BD

**T0 Read (a)**  
 $a := a - 50$   
**Write (a)**  
**Read ( b )**  
 $b := b + 50$   
**Write ( b )**

**T1 Read (a)**  
 $temp := a * 0,1$   
 $a := a - temp$   
**Write (a)**  
**Read ( b )**  
 $b := b + temp$   
**Write ( b )**

- Resolver T0, T1 o T1, T0 se respeta A+B
- Ahora bien T0 T1 <> T1 T0

4

**Planificacion:** Secuencia de ejecución de una transaccion.

Involucra todas las instrucciones de una transacción.

Conservan el orden de ejecución de las mismas.

Un conjunto de M transacciones generan m! (factorial) planificaciones en serie.

(En el ejemplo de arriba tenes 2 transacciones (2 PLANIFICACIONES), por ende  $2! = 2$  planificaciones en serie (T1-T0 Y T0-T1).

Si vos tenes un entorno concurrente, no tiene sentido ejecutar las transacciones en serie, tenes que usar la concurrencia.

En definitiva, con ejecución en serie te garantiza siempre la consistencia de la información, pero es muy lento, queremos aprovechar la concurrencia entonces está:

**Planificación concurrente:** Las instrucciones de las transacciones se van intercalando, primero unas de T0 luego otras de T1 y así siguiendo.



Acá se muestra que la ejecución concurrente de 2 transacciones SIN FALLOS EN CADA UNO, a veces genera inconsistencia y a veces no. Dependiendo de como ejecutes ambas transacciones en concurrencia te puede cambiar la consistencia de la base de datos.

**Conflictos:** SE DAN SOLO EN LOS READS Y WRITES.

Cuando 2 instrucciones operan sobre datos diferentes NO hay conflictos, es decir, no importa cual ejecute primero el resultado es el mismo y se mantiene la consistencia.

Cuando operan sobre el mismo dato:

- Si la instrucción 1 hace un READ y la instrucción 2 tmb, no importa el orden de ejecución, no genera conflictos.
- Si la instrucción 1 hace un READ y la instrucción 2 hace un Write, depende el orden de ejecución, puede generar conflictos ya que la instrucción 1 puede leer valores distintos dependiendo cuando se ejecute.
- Si la instrucción 1 hace un WRITE y la instrucción 2 hace un READ, depende el orden de ejecución, puede generar conflictos ya que la instrucción 2 puede leer valores distintos dependiendo cuando se ejecute.
- Si las 2 instrucciones hacen un WRITE depende el estado final de la BD, osea depende quien escribe primero va a hacer el valor que va a quedar. Genera conflicto creo yo.

En general, i1 e i2 entran en conflicto si actúan sobre el mismo dato y al menos una es un write.

**Definiciones:**

Una planificación S se transforma en S prima mediante intercambios de instrucciones no conflictivas (aquellas que actúan sobre datos distintos), entonces S y S prima son equivalentes en cuanto a conflictos (es decir que si en una se mantiene la consistencia, en la otra tmb, lo mismo pasa si una es inconsistente).

Para que una planificación concurrente **sea valida** yo debo poder demostrar que esta misma equivale a una planificación serie. (el dbms se debe encargar de elegir al valido).

## Metodos de control de concurrencia:

.Metodos que aseguran que la ejecución simultanea de 2 o mas transacciones no genere conflictos.

### Bloqueo: Pone de ejemplo a la impresora (para el caso exclusivo)

El uso puede ser compartido (solo lectura) o exclusivo (lectura-escritura).

Cuando un dato se bloquea de manera compartida varias transacciones pueden acceder al dato, pero solo para leer, ninguna lo puede escribir. Sin embargo, si una transaccion necesita escribir un dato, lo que se debe realizar es un bloqueo exclusivo, esto es porque yo voy a cambiar el dato, y por ende ninguna transacción lo puede leer pq yo lo estoy cambiando (si uno va a leerlo y vos lo estas cambiando puede generar conflictos) (y menos escribirlo).

Las transacciones piden lo que necesitan (recursos), usarlo y liberarlo.

Una transacción debe:

- Obtener el dato (si está libre, o compartido y solicita compartido)
- Esperar (otro caso)
- Usar el dato
- Liberarlo.

T1  $a \rightarrow b$

1. Lock\_e(a)

3. Read ( a )

4.  $a := a - 50$

5. Write (a)

6. Unlock ( a )

T2  $a + b$

2. Lock\_c(a)

7. Read ( a )

8. Unlock ( a )

9. Lock\_c(b)

11. Read ( b )

12. Unlock ( b )

17. informar (a+b)

BD A = 1000 1.exc 5. 950 6. libera 6'.comp

8. libera

B = 2000 9.comp 12. libera 12'.exclusivo

15. 2050 16. libera

T1 3. A = 1000

4. A = 950

10. bloquea!

13. B = 2000

14. B = 2050

T2 2. Bloquea!

7. A = 950

11. B = 2000

10. Lock\_e(b)

13. Read ( b )

14.  $b := b + 50$

15. Write ( b )

16. Unlock ( b )

17????????????????????

T1  $\rightarrow$  T2 o T2  $\rightarrow$  T1 en serie, no genera problemas

Hay inconsistencia en lo que informa T2, solucion: llevar los bloqueos de las transacciones al comienzo.

**Deadlock:** Yo me quedo esperando el dato B que nunca va a ser liberado porque el que lo tiene está esperando el dato A que nunca lo va a obtener porque lo tengo yo. Situación en la que una transacción espera un recurso de otra y viceversa.

Si los datos se liberan pronto, se evitan posibles deadlock.

Si los datos se mantienen bloqueados, se evita inconsistencia.

### Protocolo de bloqueos:

Dos fases. Requiere que las transacciones hagan bloqueos en 2 fases para evitar inconsistencia.

**Crecimiento:** Donde se obtienen datos (pido datos)

**Decrecimiento:** Donde se liberan datos.



Basicamente las 2 fases funcionan asi: Pedis todo lo que vas a usar y lo liberas.  
Esto no evita los deadlock.

### Basado en hora de entrada:

El orden de ejecución se determina por adelantado (no depende de quien llega primero) (el de bloqueo era asi, el q llega primero a bloquear arranca por asi decir).

Acá cada transacción recibe una hora de entrada, la cual es la hora del servidor o de un contador.

## • Si $HDE(T_i) < HDE(T_j)$ , $T_i$ es anterior

Cada tupla registra la ultima vez que se ejecutó un read o un write (la hora).

### • **Ti Solicita READ(Q)**

- $HDE(T_i) < HW(Q)$ : rechazo (solicita un dato que fue escrito por una transacción posterior)
- $HDE(T_i) \geq HW(Q)$ : ejecuta y se establece  $HR(Q) = \text{Max}\{HDE(T_i), HR(T_i)\}$

### • **Ti solicita WRITE(Q)**

- $HDE(T_i) < HR(Q)$ : rechazo (Q fue utilizado por otra transaccion anteriormente y supositu que no cambiaba)
- $HDE(T_i) < HW(Q)$ : rechazo (se intenta escribir un valor viejo, obsoleto)
- $HDE(T_i) > [HW(Q) \text{ y } HR(Q)]$ : ejecuta y  $HW(Q)$  se establece con  $HDE(T_i)$ .


- **Si Ti falla, y se rechaza entonces puede recomenzar con una nueva hora de entrada.**

20

### Si hubiera fallos: (con caso de bitácora:)

## Registro Histórico en entornos concurrentes

### Consideraciones del protocolo basado en bitácora

- Existe un único buffer de datos con  partidos y uno para la bitácora
- C/transacción tiene un área donde lleva sus datos
- El retroceso de una transacción puede llevar al retroceso de otras transacciones

### Retroceso en cascada

- Falla una transacción → pueden llevar a abortar otras
- Puede llevar a deshacer gran cantidad de trabajo.

22

Con respecto a la bitacora, en entorno concurrente es similar al monousuario salvo en los checkpoints. La idea es que pongas checkpoints cuando ninguna transacción esté activa desde ese checkpoint para arriba, pero puede que te queden transacciones activas arriba. Por ende se dispone de una lista donde te dicen todas las transacciones activas,



entonces la logica es “de donde esta el checkpoint para arriba (de las cuales ninguna de ellas estan en la lista de transacciones activas) no se las va a chequear, y luego del checkpoint para abajo si).