

# Concurrencia y Recuperabilidad

## Paradigma Pesimista

Dr. Gerardo Rossel



DEPARTAMENTO  
DE COMPUTACION

2025

## Serializabilidad

# Transacciones

## Definición básica

$$T_i \subseteq \{w_i(X); r_i(X)\} \cup \{c_i, a_i\}$$

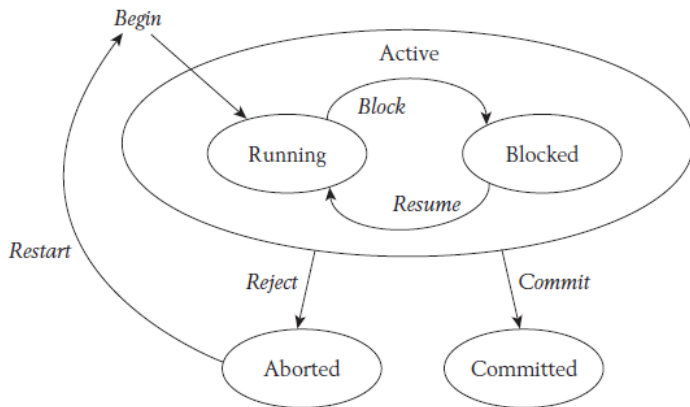
- $T_i$  representa la transacción  $i$  y  $<_i$  es el orden parcial de  $T_i$
- $w_i(X)$  escritura de la transacción  $i$  sobre el ítem  $X$
- $r_i(X)$  lectura por la transacción  $i$  del ítem  $X$
- $c_i$  representa el *commit* de la transacción  $i$
- $a_i$  representa el *abort* de la transacción  $i$
- $a_i \in T_i \iff c_i \notin T_i$
- para cualquier operación  $o_i \in T_i$  entonces  $o_i <_i a_i$  o  $o_i <_i c_i$
- si  $r_i(X), w_i(X) \in T_i$  entonces  $r_i(X) <_i w_i(X)$  o  $w_i(X) <_i r_i(X)$

## Ejemplo

$T_1 = w_1(B); r_1(C); w_1(C); c_1$

$T_2 = r_2(D); r_2(B); w_2(C); c_2$

# Estados de las transacciones



# Schedule

## Definición básica

Una *schedule* para un conjunto de ejecuciones de transacciones, es un orden parcial de las operaciones de las transacciones y que muestra cómo las transacciones son intercaladas (*interleaved*).

- Todas las operaciones en las transacciones deben aparecer en el mismo orden en el *schedule*.

El concepto de schedule provee un mecanismo para **expresarse y razonar** sobre la posible ejecución concurrente de transacciones

# Historias vs. Planes en Transacciones

## Historia (History)

- Resultado de cada transacción **ya conocido**.
- Contiene **todas las operaciones**, incluida la terminación.
- También llamadas *planes completos (complete schedules)*.

## Plan (Schedule)

- Resultado de transacciones **aún abierto**.
- Puede ser un **prefijo de una historia**: faltan operaciones o terminación.
- Representa la **planificación dinámica**: el planificador ve operaciones paso a paso.

## Sobre Historias y Schedules

Sea  $T = \{t_1, \dots, t_n\}$  un conjunto de transacciones, donde cada  $t_i \in T$  tiene la forma  $t_i = (op_i, <_i)$  con  $op_i$  denotando las operaciones de  $t_i$  y  $<_i$  su orden,  $1 \leq i \leq n$

Una **historia** para  $T$  es un par  $s = (op(s), <_s)$  tal que:

- (a)  $op(s) \subseteq \bigcup_{i=1}^n op_i \cup \bigcup_{i=1}^n \{a_i, c_i\}$
- (b) para todo  $i$ ,  $1 \leq i \leq n$ :  $c_i \in op(s) \iff a_i \notin op(s)$
- (c)  $\bigcup_{i=1}^n <_i \subseteq <_s$
- (d) para todo  $i$ ,  $1 \leq i \leq n$ , y todo  $p \in op_i$ :  $p <_s c_i$  o  $p <_s a_i$
- (e) para todo  $p, q \in op(s)$  tal que al menos uno de ellos es una escritura y ambos acceden al mismo elemento de datos:  $p <_s q$  o  $q <_s p$

Un **schedule** es un prefijo de una historia

# Equivalencia de Historias

## Operaciones Conflictivas

Dos operaciones son conflictivas si operan sobre el mismo ítem y al menos **una** de ellas es una **escritura**.



# Equivalencia de Historias

## Operaciones Conflictivas

Dos operaciones son conflictivas si operan sobre el mismo ítem y al menos **una** de ellas es una **escritura**.

## Equivalencia

Dos historias  $H_i$  y  $H_j$  son conflicto equivalentes  $H_i \equiv H_j$

- Están definidas sobre el mismo conjunto de transacciones
- El orden de las operaciones conflictivas de transacciones no abortadas es el mismo.

# Historias Serializable

## Definición

Una historia  $H$  es conflicto **serializable (SR)** si es conflicto **equivalente** a alguna historia serial  $H_s$ , es decir  $H \equiv H_s$

## CSR

Evita lost update y lecturas inconsistentes

# Testeo de Serializabilidad

## Grafo de precedencia

Se utiliza el grafo de precedencia  $SG(H)$ . Es un grafo dirigido con las siguientes características:

- Un nodo para cada transacción  $T_i \subseteq H$
- Hay ejes entre  $T_i$  y  $T_j$  sí y sólo sí hay una operación de  $T_i$  que precede en  $H$  a una operación de  $T_j$  y son operaciones conflictivas.
- Etiquetamos los ejes del grafo con los nombres de los ítems que los generan.

# Testeo de Serializabilidad

## Grafo de precedencia

Se utiliza el grafo de precedencia  $SG(H)$ . Es un grafo dirigido con las siguientes características:

- Un nodo para cada transacción  $T_i \subseteq H$
- Hay ejes entre  $T_i$  y  $T_j$  sí y sólo sí hay una operación de  $T_i$  que precede en  $H$  a una operación de  $T_j$  y son operaciones conflictivas.
- Etiquetamos los ejes del grafo con los nombres de los ítems que los generan.

## Teorema de la seriabilidad

Una historia  $H$  es **SR** sí y solo sí  $SG(H)$  es acíclico.

# Testeo de Serializabilidad - Ejemplo

## Equivalencia serial

Si el  $SG(H)$  es acíclico entonces los órdenes seriales equivalentes son los diferentes ordenes topológicos del grafo.

## Ejemplo

$$T_1 = r_1(X); w_1(X); r_1(Y); w_1(Y)$$
$$T_2 = r_2(Z); r_2(Y); w_2(Y); r_2(X); w_2(X)$$
$$T_3 = r_3(Y); r_3(Z); w_3(Y); w_3(Z)$$
$$H = r_2(Z); r_2(Y); w_2(Y); r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(X); r_1(Y); w_1(Y); w_2(X)$$

**¿Es H serializable?**

## Testeo de Serializabilidad - Ejemplo

- Dibujarlo en forma columna para visualizar mejor.

# Testeo de Serializabilidad - Ejemplo

- $H = r_2(Z); r_2(Y); w_2(Y); r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(X); r_1(Y); w_1(Y); w_2(X)$

$T_1$	$T_2$	$T_3$
	$r_2(Z)$	
	$r_2(Y)$	
	$w_2(Y)$	
		$r_3(Y)$
		$r_3(Z)$
$r_1(X)$		
$w_1(X)$		
		$w_3(Y)$
		$w_3(Z)$
	$r_2(X)$	
$r_1(Y)$		
$w_1(Y)$		
	$w_2(X)$	

## Testeo de Serializabilidad - Ejemplo

- $H = r_2(Z); r_2(Y); w_2(Y); r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(X); r_1(Y); w_1(Y); w_2(X)$

$T_1$	$T_2$	$T_3$
	$r_2(Z)$	
	$r_2(Y)$	
	$w_2(Y)$	
		$r_3(Y)$
		$r_3(Z)$
$r_1(X)$		
$w_1(X)$		
		$w_3(Y)$
		$w_3(Z)$
	$r_2(X)$	
$r_1(Y)$		
$w_1(Y)$		
	$w_2(X)$	

**T1**

**T2**

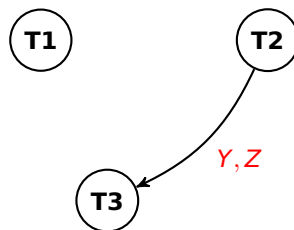
**T3**



# Testeo de Serializabilidad - Ejemplo

- $H = r_2(Z); r_2(Y); w_2(Y); r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(X); r_1(Y); w_1(Y); w_2(X)$

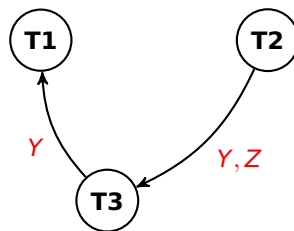
$T_1$	$T_2$	$T_3$
	$r_2(Z)$	
	$r_2(Y)$	
	$w_2(Y)$	
		$r_3(Y)$
		$r_3(Z)$
$r_1(X)$		
$w_1(X)$		
		$w_3(Y)$
		$w_3(Z)$
	$r_2(X)$	
$r_1(Y)$		
$w_1(Y)$		
	$w_2(X)$	



# Testeo de Serializabilidad - Ejemplo

- $H = r_2(Z); r_2(Y); w_2(Y); r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(X); r_1(Y); w_1(Y); w_2(X)$

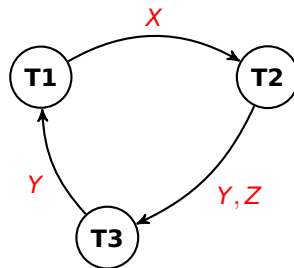
$T_1$	$T_2$	$T_3$
	$r_2(Z)$	
	$r_2(Y)$	
	$w_2(Y)$	
		$r_3(Y)$
		$r_3(Z)$
$r_1(X)$		
$w_1(X)$		
		$w_3(Y)$
		$w_3(Z)$
	$r_2(X)$	
$r_1(Y)$		
$w_1(Y)$		
	$w_2(X)$	



# Testeo de Serializabilidad - Ejemplo

- $H = r_2(Z); r_2(Y); w_2(Y); r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(X); r_1(Y); w_1(Y); w_2(X)$

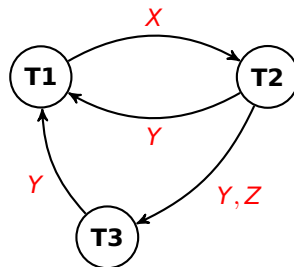
$T_1$	$T_2$	$T_3$
	$r_2(Z)$	
	$r_2(Y)$	
	$w_2(Y)$	
		$r_3(Y)$
		$r_3(Z)$
$r_1(X)$		
$w_1(X)$		
		$w_3(Y)$
		$w_3(Z)$
	$r_2(X)$	
$r_1(Y)$		
$w_1(Y)$		
	$w_2(X)$	



# Testeo de Serializabilidad - Ejemplo

- $H = r_2(Z); r_2(Y); w_2(Y); r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(X); r_1(Y); w_1(Y); w_2(X)$

$T_1$	$T_2$	$T_3$
	$r_2(Z)$	
	$r_2(Y)$	
	$w_2(Y)$	
		$r_3(Y)$
		$r_3(Z)$
$r_1(X)$		
$w_1(X)$		
		$w_3(Y)$
		$w_3(Z)$
	$r_2(X)$	
$r_1(Y)$		
$w_1(Y)$		
	$w_2(X)$	



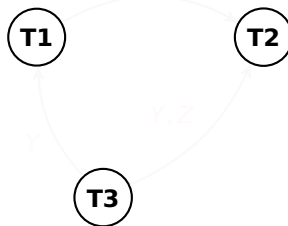
Tiene ciclos, ej:

$T_2 \rightarrow T_3 \rightarrow T_1 \rightarrow T_2$

## Testeo de Serializabilidad - Ejemplo 2

$H = r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(Z); r_1(Y); w_1(Y);$   
 $r_2(Y); w_2(Y); r_2(X); w_2(X)$

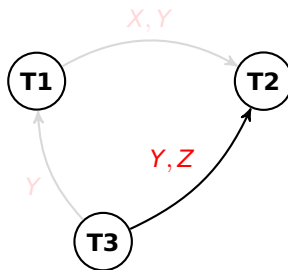
$T_1$	$T_2$	$T_3$
		$r_3(Y)$
		$r_3(Z)$
$r_1(X)$		
$w_1(X)$		
		$w_3(Y)$
		$w_3(Z)$
	$r_2(Z)$	
$r_1(Y)$		
$w_1(Y)$		
	$r_2(Y)$	
	$w_2(Y)$	
	$r_2(X)$	
	$w_2(X)$	



## Testeo de Serializabilidad - Ejemplo 2

$H = r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(Z); r_1(Y); w_1(Y);$   
 $r_2(Y); w_2(Y); r_2(X); w_2(X)$

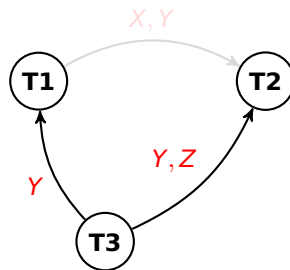
$T_1$	$T_2$	$T_3$
		$r_3(Y)$
		$r_3(Z)$
$r_1(X)$		
$w_1(X)$		
		$w_3(Y)$
		$w_3(Z)$
	$r_2(Z)$	
$r_1(Y)$		
$w_1(Y)$		
	$r_2(Y)$	
	$w_2(Y)$	
	$r_2(X)$	
	$w_2(X)$	



## Testeo de Serializabilidad - Ejemplo 2

$H = r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(Z); r_1(Y); w_1(Y);$   
 $r_2(Y); w_2(Y); r_2(X); w_2(X)$

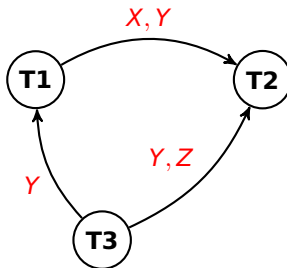
$T_1$	$T_2$	$T_3$
		$r_3(Y)$
		$r_3(Z)$
$r_1(X)$		
$w_1(X)$		
		$w_3(Y)$
		$w_3(Z)$
	$r_2(Z)$	
$r_1(Y)$		
$w_1(Y)$		
	$r_2(Y)$	
	$w_2(Y)$	
	$r_2(X)$	
	$w_2(X)$	



## Testeo de Serializabilidad - Ejemplo 2

$H = r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(Z); r_1(Y); w_1(Y);$   
 $r_2(Y); w_2(Y); r_2(X); w_2(X)$

$T_1$	$T_2$	$T_3$
		$r_3(Y)$
		$r_3(Z)$
$r_1(X)$		
$w_1(X)$		
		$w_3(Y)$
		$w_3(Z)$
	$r_2(Z)$	
$r_1(Y)$		
$w_1(Y)$		
	$r_2(Y)$	
	$w_2(Y)$	
	$r_2(X)$	
	$w_2(X)$	



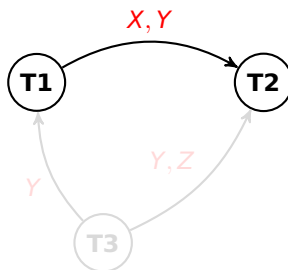
Orden Serial:



## Testeo de Serializabilidad - Ejemplo 2

$H = r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(Z); r_1(Y); w_1(Y);$   
 $r_2(Y); w_2(Y); r_2(X); w_2(X)$

$T_1$	$T_2$	$T_3$
		$r_3(Y)$
		$r_3(Z)$
$r_1(X)$		
$w_1(X)$		
		$w_3(Y)$
		$w_3(Z)$
	$r_2(Z)$	
$r_1(Y)$		
$w_1(Y)$		
	$r_2(Y)$	
	$w_2(Y)$	
	$r_2(X)$	
	$w_2(X)$	

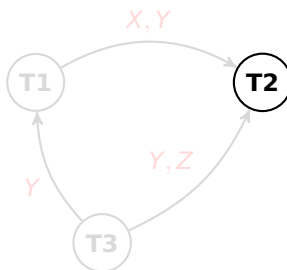


Orden Serial:  $T_3$

## Testeo de Serializabilidad - Ejemplo 2

$H = r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(Z); r_1(Y); w_1(Y);$   
 $r_2(Y); w_2(Y); r_2(X); w_2(X)$

$T_1$	$T_2$	$T_3$
		$r_3(Y)$
		$r_3(Z)$
$r_1(X)$		
$w_1(X)$		
		$w_3(Y)$
		$w_3(Z)$
	$r_2(Z)$	
$r_1(Y)$		
$w_1(Y)$		
	$r_2(Y)$	
	$w_2(Y)$	
	$r_2(X)$	
	$w_2(X)$	

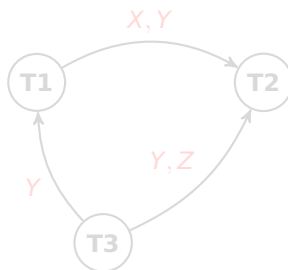


Orden Serial:  $T_3 \rightarrow T_1$

## Testeo de Serializabilidad - Ejemplo 2

$H = r_3(Y); r_3(Z); r_1(X); w_1(X); w_3(Y); w_3(Z); r_2(Z); r_1(Y); w_1(Y);$   
 $r_2(Y); w_2(Y); r_2(X); w_2(X)$

$T_1$	$T_2$	$T_3$
		$r_3(Y)$
		$r_3(Z)$
$r_1(X)$		
$w_1(X)$		
		$w_3(Y)$
		$w_3(Z)$
	$r_2(Z)$	
$r_1(Y)$		
$w_1(Y)$		
	$r_2(Y)$	
	$w_2(Y)$	
	$r_2(X)$	
	$w_2(X)$	



Orden Serial:  $T_3 \rightarrow T_1 \rightarrow T_2$

# Order preserving conflict serializability

## Definición

Una historia  $s$  se llama **order-preserving conflict serializable (OCSR)** si:

- 1 Es **conflicto-serializable**, es decir, existe una historia serial  $s'$  tal que  $op(s) = op(s')$  y  $s \approx_c s'$ .
- 2 Para todas las transacciones  $t, t' \in trans(s)$ : si  $t$  ocurre completamente antes que  $t'$  en  $s$ , entonces lo mismo ocurre en  $s'$ .

## Notación

Denotamos por **OCSR** la clase de todas las historias conflicto-serializables que preservan el orden.

## Recuperabilidad

## Problemas de recuperabilidad

### Recuperación

Se puede asumir que el *Abort* se implementa recuperando imágenes anteriores de los ítems.

# Problemas de recuperabilidad

## Recuperación

Se puede asumir que el *Abort* se implementa recuperando imágenes anteriores de los ítems.

## Ejemplos

- $H_1 = w_1(X, 2); r_2(X); w_2(Y, 3); c_2.$ 
  - Si  $T_1$  aborta deberíamos abortar  $T_2$  (violaríamos la semántica del *commit*)

# Problemas de recuperabilidad

## Recuperación

Se puede asumir que el *Abort* se implementa recuperando imágenes anteriores de los ítems.

## Ejemplos

- $H_1 = w_1(X, 2); r_2(X); w_2(Y, 3); c_2.$ 
  - Si  $T_1$  aborta deberíamos abortar  $T_2$  (violaríamos la semántica del *commit*)
- $H_1 = w_1(X); r_2(X); w_2(Y); a_1.$ 
  - Aborts en cascada



# Problemas de recuperabilidad

## Recuperación

Se puede asumir que el *Abort* se implementa recuperando imágenes anteriores de los ítems.

## Ejemplos

- $H_1 = w_1(X, 2); r_2(X); w_2(Y, 3); c_2.$ 
  - Si  $T_1$  aborta deberíamos abortar  $T_2$  (violaríamos la semántica del *commit*)
- $H_1 = w_1(X); r_2(X); w_2(Y); a_1.$ 
  - Aborts en cascada
- $H_1 = w_1(X); w_2(X); a_1; a_2.$ 
  - Problema para recuperar imagen

$w_1(X, 2)$  significa que la transacción 1 escribe el valor 2 en X

# Lost Update

$t_1$	Time	$t_2$
	/* $x = 100$ */	
$r(x)$	1	
	2	$r(x)$
/* update $x := x + 30$ */	3	
	4	/* update $x := x + 20$ */
$w(x)$	5	
	/* $x = 130$ */	
	6	$w(x)$
	/* $x = 120$ */	
↑ update "lost"		

X debería tener 150, pero se pierde la actualización de  $t_1$

# Inconsistent-read

Inicialmente  $x = y = 50$  entonces  $x + y = 100$

$t_1$	Time	$t_2$
	1	$r(x)$
	2	$/* x := x - 10 */$
	3	$w(x)$
$/* sum := 0 */$	4	
$r(x)$	5	
$r(y)$	6	
$/* sum := sum + x */$	7	
$/* sum := sum + y */$	8	
	9	$r(y)$
	10	$/* y := y + 10 */$
	11	$w(y)$

# Dirty-read

$t_1$	Time	$t_2$
$r(x)$	1	
$/* x := x + 100 */$	2	
$w(x)$	3	
	4	$r(x)$
	5	$/* x := x - 100 */$
failure & rollback	6	
	7	$w(x)$

# Lectura entre transacciones

Dadas dos transacciones  $T_i$  y  $T_j$  decimos que  $T_i$  lee  $X$  de  $T_j$  si  $T_i$  lee  $X$  y  $T_j$  fue la última transacción que escribió  $X$  y no abortó antes de que  $T_i$  lo leyera.

- 1  $w_j(X) < r_i(X)$
- 2  $a_j \not< r_i(X)$
- 3 Si hay algún  $w_k(X)$  tal que  $w_j(X) < w_k(X) < r_i(X)$  entonces  $a_k < r_i(X)$

# Niveles de recuperabilidad

## Historia Recuperable **RC**

Una historia  $H$  es **RC** si siempre que una transacción  $T_i$  lee de  $T_j$  con  $i \neq j$  en  $H$  y  $c_i \in H$  entonces  $c_j < c_i$ .

Intuitivamente una historia es recuperable si una transacción realiza commit sólo después de que hicieron commit todas las transacciones de las cuales lee.

# Niveles de recuperabilidad

## Historia Recuperable **RC**

Una historia  $H$  es **RC** si siempre que una transacción  $T_i$  lee de  $T_j$  con  $i \neq j$  en  $H$  y  $c_i \in H$  entonces  $c_j < c_i$ .

Intuitivamente una historia es recuperable si una transacción realiza commit sólo después de que hicieron commit todas las transacciones de las cuales lee.

## Avoids Cascading Aborts **ACA**

Una historia  $H$  es **ACA** si siempre que una transacción  $T_i$  lee  $X$  de  $T_j$  con  $i \neq j$  en  $H$  entonces  $c_j < r_i(X)$ .

Lee sólo valores de transacciones que ya hicieron *commit*

# Niveles de recuperabilidad

## Historia Recuperable **RC**

Una historia  $H$  es **RC** si siempre que una transacción  $T_i$  lee de  $T_j$  con  $i \neq j$  en  $H$  y  $c_i \in H$  entonces  $c_j < c_i$ .

Intuitivamente una historia es recuperable si una transacción realiza commit sólo después de que hicieron commit todas las transacciones de las cuales lee.

## Avoids Cascading Aborts **ACA**

Una historia  $H$  es **ACA** si siempre que una transacción  $T_i$  lee  $X$  de  $T_j$  con  $i \neq j$  en  $H$  entonces  $c_j < r_i(X)$ .

Lee sólo valores de transacciones que ya hicieron *commit*

## Stricta **ST**

Una historia  $H$  es **ST** si siempre que  $w_j(X) < o_i(X)$  con  $i \neq j$  entonces  $a_j < o_i(X)$  o  $c_j < o_i(X)$  siendo  $o_i(X)$  igual a  $r_i(X)$  o a  $w_i(X)$

Es decir no se puede leer ni escribir un ítem hasta que la transacción que lo escribió previamente haya hecho *commit* o *abort*.



## Teorema de la recuperabilidad

### Teorema

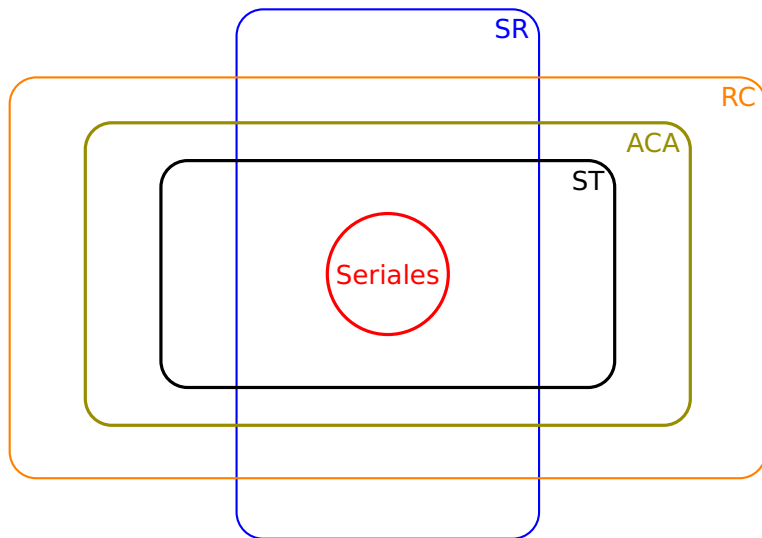
$$ST \subset ACA \subset RC$$

### Ortogonalidad

SR intersecta a todos los conjuntos RC, ACA y ST. Son conceptos ortogonales.

Es fácil ver que una historia serial es también ST.

## Relación entre recuperabilidad y serializabilidad



# Rigurosidad RG

## Definición de Schedule RigurosoRG

Un schedule  $s$  es riguroso (**RG**) si es estricto y además satisface las siguiente condición de rigurosidad:

$\forall t_i, t_j \in trans(s)$ , Si  $r_j(x) <_s w_i(x)$ ,  $i \neq j$  entonces  
 $a_j <_s w_i(x) \vee c_j <_s w_i(x)$

## ATENCION

La característica notable de RG es que a diferencia de las otras tipos de recuperabilidad (RC, ACA y ST) no es ortogonal con respecto a la serializabilidad, sino que por el contrario

$RG \subset SR$

Además sabemos que  $RG \subset ST \subset ACA \subset RC$

# Propiedad de Prefijos Commit en Clases de Schedules

## Definición

Una propiedad que se puede verificar fácilmente usando las definiciones de **RG**, **ST**, **ACA** y **RC** es que la pertenencia a cualquiera de estas cuatro clases es **cerrada bajo prefijos commit**.

## Interpretación

- Si un schedule pertenece, por ejemplo, a **ACA**, la misma propiedad se cumple para la **proyección de los commits** de cada uno de sus prefijos.
- Esto permite analizar schedules de manera incremental, mientras las operaciones van llegando.
- Facilita la verificación de consistencia y control de concurrencia en tiempo real.

## Control de Concurrencia

# Utilización de locks

## Lock

Un **lock** es una variable asociada con un ítem de datos que describe el estado de ese ítem con respecto a posibles operaciones que pueden aplicarse a él.

## Problemas

- Deadlock
- Livelocks

# Lock o Bloqueo Binario

## Lock Binario

Un **lock binario** es el modelo más simple. No es utilizado en las BD reales, pero es útil para comenzar y luego pasar a modelos más realistas.

**Locks binarios** pueden tener uno de dos estados: *locked* o *unlocked*

## Notación

- $l_i(A)$  **Lock**. La transacción  $i$  realiza un *bloqueo o lock* sobre el ítem  $A$ .
- $u_i(A)$  **UnLock**. La transacción  $i$  libera los *bloqueos o locks* previos sobre el ítem  $A$ . Usado en todos los modelos, se asume que libera todos los *locks* tomados.

# Lock o Bloqueo Binario

## Lock binario

El **lock binario** fuerza exclusión mutua sobre un ítem  $X$ .

Las transacciones pueden ser vistas como una secuencia de locks y unlocks

## Historias Legales

Consistencia de Transacciones:

- 1 Una  $T_i$  puede leer o escribir un ítem  $X$  si previamente realizó un lock sobre  $X$  y no lo ha liberado
- 2 Si una transacción  $T_i$  realiza un lock sobre un elemento debe posteriormente liberarlo.

Legalidad:

- Una  $T_i$  que desea obtener un lock sobre  $X$  que ha sido lockeado por  $T_j$  en un modo que conflictua, debe esperar hasta que  $T_j$  haga unlock de  $X$ .



# Grafo de precedencia para lock binario

Se asume  $H$  legal. Para hacer el  $SG(H)$  se siguen los siguientes pasos:

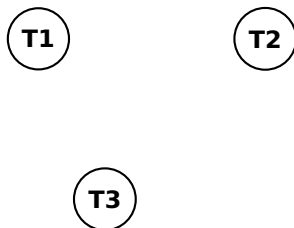
- 1 Hacer un nodo por cada  $T_i \subseteq H$
- 2 Si  $T_i$  realiza un  $l_i(X)$  para algún ítem  $X$  y luego  $T_j$  con  $i \neq j$  realiza un  $l_i(X)$  hacer un arco  $T_i \rightarrow T_j$

# Grafo de precedencia para lock binario

Se asume  $H$  legal. Para hacer el  $SG(H)$  se siguen los siguientes pasos:

- 1 Hacer un nodo por cada  $T_i \subseteq H$
- 2 Si  $T_i$  realiza un  $l_i(X)$  para algún ítem  $X$  y luego  $T_j$  con  $i \neq j$  realiza un  $l_i(X)$  hacer un arco  $T_i \rightarrow T_j$

Ejemplo:  $H = l_2(A); u_2(A); l_3(A); u_3(A); l_1(B); u_1(B); l_2(B); u_2(B)$

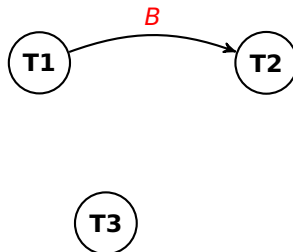


# Grafo de precedencia para lock binario

Se asume  $H$  legal. Para hacer el  $SG(H)$  se siguen los siguientes pasos:

- 1 Hacer un nodo por cada  $T_i \subseteq H$
- 2 Si  $T_i$  realiza un  $l_i(X)$  para algún ítem  $X$  y luego  $T_j$  con  $i \neq j$  realiza un  $l_i(X)$  hacer un arco  $T_i \rightarrow T_j$

Ejemplo:  $H = l_2(A); u_2(A); l_3(A); u_3(A); l_1(B); u_1(B); l_2(B); u_2(B)$

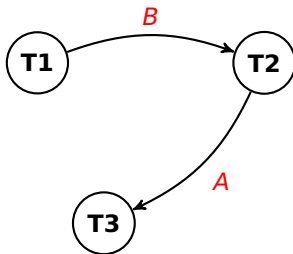


# Grafo de precedencia para lock binario

Se asume  $H$  legal. Para hacer el  $SG(H)$  se siguen los siguientes pasos:

- 1 Hacer un nodo por cada  $T_i \subseteq H$
- 2 Si  $T_i$  realiza un  $l_i(X)$  para algún ítem  $X$  y luego  $T_j$  con  $i \neq j$  realiza un  $l_i(X)$  hacer un arco  $T_i \rightarrow T_j$

Ejemplo:  $H = l_2(A); u_2(A); l_3(A); u_3(A); l_1(B); u_1(B); l_2(B); u_2(B)$

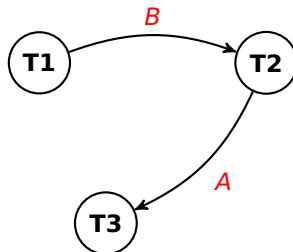


# Grafo de precedencia para lock binario

Se asume  $H$  legal. Para hacer el  $SG(H)$  se siguen los siguientes pasos:

- 1 Hacer un nodo por cada  $T_i \subseteq H$
- 2 Si  $T_i$  realiza un  $l_i(X)$  para algún ítem  $X$  y luego  $T_j$  con  $i \neq j$  realiza un  $l_i(X)$  hacer un arco  $T_i \rightarrow T_j$

Ejemplo:  $H = l_2(A); u_2(A); l_3(A); u_3(A); l_1(B); u_1(B); l_2(B); u_2(B)$



$H$  es SR y la Historia Serial Equivalente es  $T_1, T_2, T_3$

# Lock Ternario

## Motivación

Debido a que operaciones de lectura de diferentes transacciones sobre el mismo ítem no son conflictivas se puede permitir que accedan sólo para lectura.

## Atención

Sin embargo si una transacción desea escribir debe tener acceso exclusivo al ítem.

# Lock Ternario

## Motivación

Debido a que operaciones de lectura de diferentes transacciones sobre el mismo ítem no son conflictivas se puede permitir que accedan sólo para lectura.

## Atención

Sin embargo si una transacción desea escribir debe tener acceso exclusivo al ítem.

## 2 tipos de locks

$rl_i(A)$  **Lock de lectura o compartido**. La transacción  $i$  realiza un *bloqueo o lock* de lectura sobre el ítem  $A$ .

$wl_i(A)$  **Lock de escritura o exclusivo**. La transacción  $i$  realiza un *lock* exclusivo o de escritura sobre el ítem  $A$ .

# Matriz de Compatibilidad

		Lock Sostenido por $T_j$	
		<b>rl</b>	<b>wl</b>
Lock solicitado por $T_i$	<b>rl</b>	Si	No
	<b>wl</b>	No	No



# Legalidad y Consistencia

## Consistencia

- (a) Una acción  $r_i(X)$  debe ser precedida por un  $rl_i(X)$  o un  $wl_i(X)$ , sin que intervenga un  $u_i(X)$
- (b) Una acción  $w_i(X)$  debe ser precedida por una  $wl_i(X)$  sin que intervenga un  $u_i(X)$
- (c) Todos los *locks* deben ser seguidos de un *unlock* del mismo elemento

## Legalidad de las Historias

- (a) Si  $wl_i(X)$  aparece en una historia, entonces no puede haber luego un  $wl_j(X)$  o  $rl_j(X)$  para  $j \neq i$  sin que haya primero un  $u_i(X)$
- (b) Si  $rl_i(X)$  aparece en una historia no puede haber luego un  $wl_j(X)$  para  $j \neq i$  sin que haya primero un  $u_i(X)$

# Grafo de precedencia para Locking ternario

- 1 Hacer un nodo por cada  $T_i$
- 2 Si  $T_i$  hace un  $rl_i(X)$  o  $wl_i(X)$  y luego  $T_j$  con  $j \neq i$  hace un  $wl_j(X)$  en  $H$  hacer un arco  $T_i \rightarrow T_j$
- 3 Si  $T_i$  hace un  $wl_i(X)$  y  $T_j$  con  $j \neq i$  hace un  $rl_j(X)$  en  $H$  entonces hacer un arco  $T_i \rightarrow T_j$

Básicamente dice que si dos transacciones realizan un *lock* sobre el mismo ítem y al menos uno de ellas es un *write lock* se debe dibujar un eje desde la primera a la segunda.

## Conversión o Upgrading/Downgrading Lock

### Conversión

Una transacción que tiene un *lock* sobre un ítem *X* tiene permitido bajo ciertas condiciones convertir dicho *lock* en otro tipo de *lock*.

La forma más común es el *upgrading lock*, es decir pasar de un *lock de escritura o compartido* **a un lock exclusivo o de escritura**.

# Upgrade Lock y Update Lock

## Deadlock al usar upgrade lock

Supongamos  $T_1$  y  $T_2$ , y se presenta la siguiente historia donde cada una quiere realizar un upgrade lock. Ambos son denegados:

$H = rl_1(X); rl_2(X); wl_1(X); wl_2(X)^a$

---

<sup>a</sup> Las operaciones en rojo indican que no pudieron ser completadas y deben esperar

## Update Lock

Se puede evitar este problema del deadlock si agregamos otro modo de *lock* llamado **update lock**. Un update lock sobre un ítem  $X$  que denotamos  $ul_i(X)$  da a la transacción  $T_i$  privilegio de lectura sobre  $X$  pero no de escritura. Como ventaja el *update lock* pasa a ser el **único** que puede ser *upgraded* a *write lock*

# Matriz de compatibilidad Lock

## Matriz de compatibilidad Update Lock

		Lock Sostenido		
		rl	wl	ul
Lock Solicitado	rl	Si	No	No
	wl	No	No	No
	ul	Si	No	No

## Uso de update lock

$H = rl_1(X); rl_2(X); wl_1(X); wl_2(X)$  Deadlock

$T_1 = ul_1(X); wl_1(X); u_1(X)$

$T_2 = ul_2(X); wl_2(X); u_2(X)$

$H = ul_1(X); ul_2(X); wl_1(X); u_1(X); ul_2(X); wl_2(X); u_2(X)$

# Loking Y Serializabilidad

## Atención

El mecanismo de locking por si solo no garantiza serializabilidad. Se necesita utilizar un protocolo para posicionar los locks y unlocks.

# Loking Y Serializabilidad

## Atención

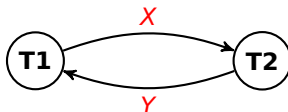
El mecanismo de locking por si solo no garantiza serializabilidad. Se necesita utilizar un protocolo para posicionar los locks y unlocks.

$$T_1 = rl_1(X); u_1(X); wl_1(Y); u_1(Y); c_1$$
$$T_2 = wl_2(X); wl_2(Y); u_2(X); u_2(Y); c_2$$
$$H = rl_1(X); u_1(X); wl_2(X); wl_2(Y); u_2(X); u_2(Y); c_2; wl_1(Y); u_1(Y); c_1$$

# Loking Y Serializabilidad

## Atención

El mecanismo de locking por si solo no garantiza serializabilidad. Se necesita utilizar un protocolo para posicionar los locks y unlocks.

$$T_1 = rl_1(X); u_1(X); wl_1(Y); u_1(Y); c_1$$
$$T_2 = wl_2(X); wl_2(Y); u_2(X); u_2(Y); c_2$$
$$H = rl_1(X); u_1(X); wl_2(X); wl_2(Y); u_2(X); u_2(Y); c_2; wl_1(Y); u_1(Y); c_1$$




## Two Phase Locking - 2PL

### Two Phase Locking - Definición

Una transacción respeta el protocolo de bloqueo en dos fases (2PL) si todas las operaciones de bloqueo (*lock*) preceden a la primer operación de desbloqueo (*unlock*) en la transacción.

Una transacción que cumple con el protocolo se dice que es una **transacción 2PL**

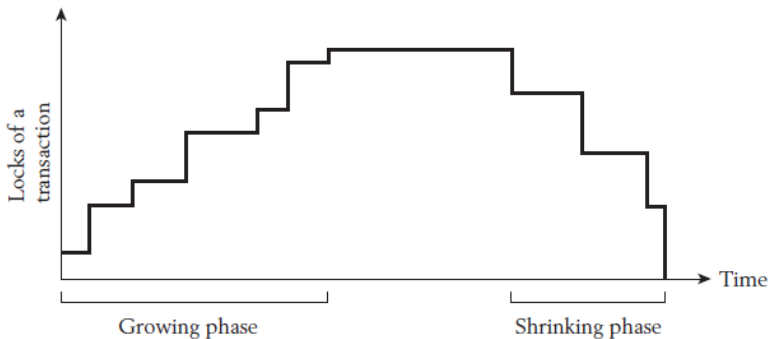
- Fase de crecimiento: toma los locks
- Fase de contracción: libera los locks

### Serializabilidad con 2PL

Dado  $T = T_1, T_2, \dots, T_n$ , si toda  $T_i$  en  $T$  es **2PL**, entonces todo  $H$  legal sobre  $T$  es **SR**.

Es mas se cumple que también  $Gen(2PL) \subset OCSR$

# 2PL

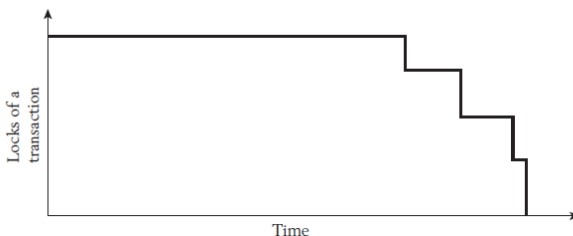


Ojo puede una historia ser 2PL y no ser ACA.

# Variantes de 2PL

## 2PL Conservador (C2PL)

Bajo 2PL estático o conservador (C2PL), cada transacción adquiere todos los locks que necesitará al inicio, es decir, antes de ejecutar su primer paso de lectura (r) o escritura (w). Esto también se conoce como pre-reclamar todos los locks necesarios de antemano



## Variantes de 2PL

### 2PL Estricto (2PLE o *S2PL*)

Una transacción cumple con **2PL Estricto** si es 2PL y no libera ninguno de sus **locks de escritura** hasta **después** de realizar el *commit* o el *abort*. 2PLE garantiza que la historia es **ST**.

### 2PL Riguroso (2PLR o *SS2PL*)

Una transacción cumple con **2PL Riguroso** si es 2PL y no libera ninguno de sus **locks de escritura o lectura** hasta **después** de realizar el *commit* o el *abort*.

# Deadlocks

## Definición

Deadlock es un estado en el cual cada miembro de un grupo de transacciones está esperando que algún otro miembro libere un *lock*.

## ¿Cómo tratar con Deadlocks?

- Prevención
- Detección

## Detección usando Wait-for Graph

- Un nodo por cada transacción que tiene un lock o espera por uno.
- Un eje entre dos nodos ( $T_i$  y  $T_j$ ) si  $T_i$  está esperando que  $T_j$  libere un lock que sobre un ítem que  $T_i$  necesita bloquear.

Considerar la siguiente historia parcial:

$I_1(A); I_2(B); I_1(B); I_3(C); I_2(C); I_4(B); I_3(A)$

# Detección usando Wait-for Graph

- Un nodo por cada transacción que tiene un lock o espera por uno.
- Un eje entre dos nodos ( $T_i$  y  $T_j$ ) si  $T_i$  está esperando que  $T_j$  libere un lock que sobre un ítem que  $T_i$  necesita bloquear.

Considerar la siguiente historia parcial:

$l_1(A); l_2(B); l_1(B); l_3(C); l_2(C); l_4(B); l_3(A)$



# Detección usando Wait-for Graph

- Un nodo por cada transacción que tiene un lock o espera por uno.
- Un eje entre dos nodos ( $T_i$  y  $T_j$ ) si  $T_i$  está esperando que  $T_j$  libere un lock que sobre un ítem que  $T_i$  necesita bloquear.

Considerar la siguiente historia parcial:

$l_1(A); l_2(B); l_1(B); l_3(C); l_2(C); l_4(B); l_3(A)$





## Detección usando Wait-for Graph

- Un nodo por cada transacción que tiene un lock o espera por uno.
- Un eje entre dos nodos ( $T_i$  y  $T_j$ ) si  $T_i$  está esperando que  $T_j$  libere un lock que sobre un ítem que  $T_i$  necesita bloquear.

Considerar la siguiente historia parcial:

$I_1(A); I_2(B); I_1(B); I_3(C); I_2(C); I_4(B); I_3(A)$

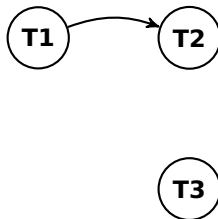


# Detección usando Wait-for Graph

- Un nodo por cada transacción que tiene un lock o espera por uno.
- Un eje entre dos nodos ( $T_i$  y  $T_j$ ) si  $T_i$  está esperando que  $T_j$  libere un lock que sobre un ítem que  $T_i$  necesita bloquear.

Considerar la siguiente historia parcial:

$I_1(A); I_2(B); I_1(B); I_3(C); I_2(C); I_4(B); I_3(A)$

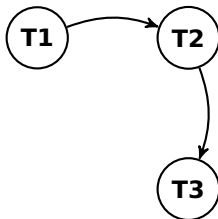


# Detección usando Wait-for Graph

- Un nodo por cada transacción que tiene un lock o espera por uno.
- Un eje entre dos nodos ( $T_i$  y  $T_j$ ) si  $T_i$  está esperando que  $T_j$  libere un lock que sobre un ítem que  $T_i$  necesita bloquear.

Considerar la siguiente historia parcial:

$I_1(A); I_2(B); I_1(B); I_3(C); I_2(C); I_4(B); I_3(A)$

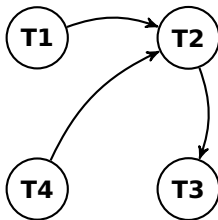


## Detección usando Wait-for Graph

- Un nodo por cada transacción que tiene un lock o espera por uno.
- Un eje entre dos nodos ( $T_i$  y  $T_j$ ) si  $T_i$  está esperando que  $T_j$  libere un lock que sobre un ítem que  $T_i$  necesita bloquear.

Considerar la siguiente historia parcial:

$I_1(A); I_2(B); I_1(B); I_3(C); I_2(C); I_4(B); I_3(A)$

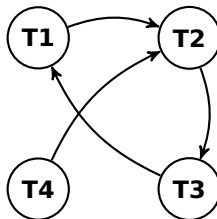


# Detección usando Wait-for Graph

- Un nodo por cada transacción que tiene un lock o espera por uno.
- Un eje entre dos nodos ( $T_i$  y  $T_j$ ) si  $T_i$  está esperando que  $T_j$  libere un lock que sobre un ítem que  $T_i$  necesita bloquear.

Considerar la siguiente historia parcial:

$I_1(A); I_2(B); I_1(B); I_3(C); I_2(C); I_4(B); I_3(A)$



## Elección de víctima.

- Cuanto tiempo la transacción ha estado ejecutándose. Sería mejor abortar una transacción más joven que una que ha estado ejecutándose por más tiempo
- Cuantos ítems de datos han sido actualizados por la transacción. Sería mejor abortar una transacción que hizo pocas modificaciones a la base de datos. Es decir que tiene la menor cantidad de registros de log.
- Cuantos ítems de datos le faltan actualizar. Aunque esto puede ser algo que el DBMS no necesariamente sepa.
- El número de ciclos que contiene la transacción. Mientras más ciclos tenga mejor es.

### Configurable

En Microsoft SQL Server, una transacción puede configurarse como:  
"SET DEADLOCK\_PRIORITY LOW" or "SET DEADLOCK\_PRIORITY NORMAL."

# Prevención usando TimeStamp

## TimeStamp

Cada transacción  $T_i$  recibe un timestamp  $TS(T_i)$ . Es un identificador único basado en el orden en el cual cada transacción comienza. Si  $TS(T_i) < TS(T_j)$  significa que  $T_i$  es más vieja que  $T_j$

# Prevención usando TimeStamp

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

- **Wait-Die**

- Si  $TS(T_i) < TS(T_j)$  ( $T_i$  más viejo que  $T_j$ ), entonces  $T_i$  se lo pone en espera,
- Si  $TS(T_i) > TS(T_j)$  ( $T_i$  más joven que  $T_j$ ), entonces se aborta  $T_i$  ( $T_i$  *dies*) y se recomienza mas tarde con el mismo timestamp.

- **Wound-Wait**

- Si  $TS(T_i) < TS(T_j)$  ( $T_i$  más viejo que  $T_j$ ), entonces, abortar  $T_j$  ( $T_i$  *wounds*  $T_j$ ) y recomienza más tarde con el mismo timestamp.
- Si  $TS(T_i) > TS(T_j)$  ( $T_i$  más joven que  $T_j$ ), entonces,  $T_i$  se pone en espera.



## Otros esquemas

- Timeout
- No waiting (NW)
- Cautious waiting (CW)
  - Cuando  $T_i$  quiere bloquear un ítem que está bloqueado por  $T_j$ : Si  $T_j$  no está bloqueada (no esta esperando por algún otro ítem bloqueado) entonces  $T_i$  es bloqueado y espera. En otro caso  $T_i$  aborta

## Bibliografía

- **Concurrency Control and Recovery in Database -**  
Addison Wesley – 1987 (by Philip Bernstein, Vassos Radzilacos, Vassos Hadzilacos)
- **Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery** Gerhard Weikum and Gottfried Vossen. 2001. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
- **Database Systems: The Complete Book**, Prentice Hall, 2nd Edition - Prentice Hall - 2009 (by Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom )