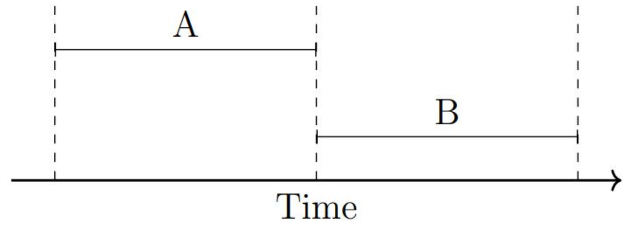
# Control de la concurrencia

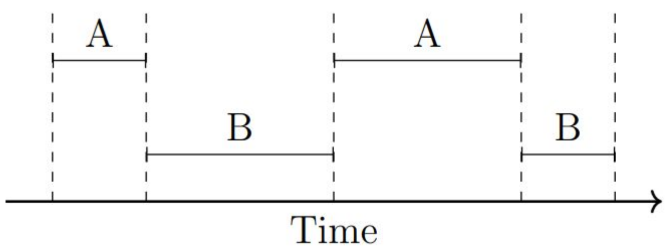
Sivana Hamer - sivana.hamer@ucr.ac.cr
Escuela de Ciencias de la Computación


En sistemas mono-usuario, se puede ejecutar todo dentro de un thread serialmente





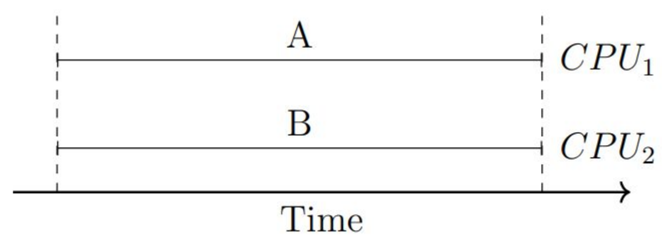

También se puede ejecutar intercaladamente un proceso




(b) Interleaved processing



Se puede procesar totalmente paralelo en varias CPUs




(c) Parallel processing

## Un *schedule* es el orden en que se ejecutan las transacciones

| $T_1$ | $T_2$ |
|---|---|
| read$(A)$; | |
| $A := A - 100$; | |
| write$(A)$; | |
| read$(B)$; | |
| $B := B + 100$; | |
| write$(B)$; | |
| commit; | |
| | read$(A)$; |
| | $A := A + 50$; |
| | write$(A)$; |
| | commit; |



## Se puede tener un *schedule serial*

| $T_1$ | $T_2$ |
|---|---|
| | read$(A)$; |
| | $A := A + 50$; |
| | write$(A)$; |
| | commit; |
| read$(A)$; | |
| $A := A - 100$; | |
| write$(A)$; | |
| read$(B)$; | |
| $B := B + 100$; | |
| write$(B)$; | |
| commit; | |



## Se puede tener un *schedule* equivalente a ejecución serial

| $T_1$ | $T_2$ |
|---|---|
| read$(A)$; | |
| $A := A - 100$; | |
| write$(A)$; | |
| | read$(A)$; |
| | $A := A + 50$; |
| | write$(A)$; |
| | commit; |
| read$(B)$; | |
| $B := B + 100$; | |
| write$(B)$; | |
| commit; | |



## Se puede tener un *schedule* que deja un estado inconsistente

| $T_1$ | $T_2$ |
|---|---|
| read$(A)$; | |
| $A := A - 100$; | |
| | read$(A)$; |
| | $A := A + 50$; |
| | write$(A)$; |
| | commit; |
| write$(A)$; | |
| read$(B)$; | |
| $B := B + 100$; | |
| write$(B)$; | |
| commit; | |

Con la ejecución concurrente, pueden suceder varios problemas



| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| | read($A$); |
| | $A := A + 50$; |
| **write($A$);** | |
| | **write($A$);** |
| read($B$); | |
| $A := B + 100$; | |
| write($B$); | |

Figure 8: Lost update example



| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| **write($A$);** | |
| | **read($A$);** |
| | $A := A + 50$; |
| | write($A$); |
| read($B$); | |
| ABORT; | |



| $T_1$ | $T_2$ |
|---|---|
| **read($A$);** | |
| $A := A - 50$; | |
| write($A$); | |
| | read($A$); |
| | $A := A + 50$; |
| | **write($A$);** |
| **read($A$);** | |
| $A := A - 50$; | |
| write($A$); | |

Figure 10: Unrepeatable read example

| $T_1$ | $T_2$ |
|---|---|
| **read**$(tA)$; | |
| | **insert**$(a_{n+1}$ **in** $tA)$; |
| **read**$(tA)$; | |

Figure 11: Phantom read example



La seriabilidad determina si una ejecución concurrente es equivalente a una ejecución serial

| $T_1$ | $T_2$ |
|---|---|
| read$(A)$;<br>$A := A - 100$;<br>write$(A)$;<br>read$(B)$;<br>$B := B + 100$;<br>write$(B)$;<br>commit; | |
| | read$(A)$;<br>$A := A + 50$;<br>write$(A)$;<br>commit; |

≡

| $T_1$ | $T_2$ |
|---|---|
| read$(A)$;<br>$A := A - 100$;<br>write$(A)$; | |
| | read$(A)$;<br>$A := A + 50$;<br>write$(A)$;<br>commit; |
| read$(B)$;<br>$B := B + 100$;<br>write$(B)$;<br>commit; | |



Existen dos tipos de seriabilidad…

- Basado en conflictos
- Basado en vistas



Se pueden generar conflictos por el mismo ítem de datos cuando…

| | **Write** | **Read** |
|---|---|---|
| **Write** | ✓ | ✓ |
| **Read** | ✓ | |

If $I$ and $J$ are two consecutive instructions of a schedule $S$ that do not have a conflict, then we can swap the order of $I$ and $J$ to generate a new schedule for $S'$. $S$ is equivalent to $S'$ as all instructions have the same order except for $I$ and $J$ whose order does not matter. If schedule $S$ can be transformed into a schedule $S'$ by swapping non-conflicting instructions, then $S$ and $S'$ are *conflict equivalent*.



| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| | write($A$); |
| read($B$); | |
| write($B$); | |

(a) Concurrent schedule



| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| | write($A$); |
| read($B$); | |
| write($B$); | |

(a) Concurrent schedule

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| **read($B$);** | |
| | **write($A$);** |
| write($B$); | |

(b) $T_1$ *read*($B$) $\leftrightarrow$ $T_2$ *write*($A$)



| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| | write($A$); |
| read($B$); | |
| write($B$); | |

(a) Concurrent schedule

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| **read($B$);** | |
| | **write($A$);** |
| write($B$); | |

(b) $T_1$ *read*($B$) $\leftrightarrow$ $T_2$ *write*($A$)

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| read($B$); | |
| **write($B$);** | |
| | **write($A$);** |

(c) $T_1$ *write*($B$) $\leftrightarrow$ $T_2$ *write*($A$)

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| | write($A$); |
| read($B$); | |
| write($B$); | |

(a) Concurrent schedule

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| **read($B$);** | |
| | **write($A$);** |
| write($B$); | |

(b) $T_1$ $read(B) \leftrightarrow T_2$ $write(A)$

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| read($B$); | |
| **write($B$);** | |
| | **write($A$);** |

(c) $T_1$ $write(B) \leftrightarrow T_2$ $write(A)$

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| **read($B$);** | |
| | **read($A$);** |
| write($B$); | |
| | write($A$); |

(d) $T_1$ $read(B) \leftrightarrow T_2$ $read(A)$



| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| | write($A$); |
| read($B$); | |
| write($B$); | |

(a) Concurrent schedule

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| **read($B$);** | |
| | **write($A$);** |
| write($B$); | |

(b) $T_1$ $read(B) \leftrightarrow T_2$ $write(A)$

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| | read($A$); |
| read($B$); | |
| **write($B$);** | |
| | **write($A$);** |

(c) $T_1$ $write(B) \leftrightarrow T_2$ $write(A)$

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| **read($B$);** | |
| | **read($A$);** |
| write($B$); | |
| | write($A$); |

(d) $T_1$ $read(B) \leftrightarrow T_2$ $read(A)$

| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| write($A$); | |
| read($B$); | |
| **write($B$);** | |
| | **read($A$);** |
| | write($A$); |

(e) $T_1$ $read(B) \leftrightarrow T_2$ $read(A)$



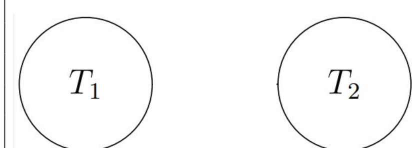
También se puede generar un gráfico de dependencia con las siguientes condiciones…

- $T_i$ executes a $write(Q)$ before $T_j$ executes a $read(Q)$.
- $T_i$ executes a $read(Q)$ before $T_j$ executes a $write(Q)$.
- $T_i$ executes a $write(Q)$ before $T_j$ executes a $write(Q)$.

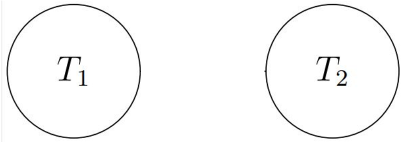
edge $T_i$ to $T_j$



| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| write($A$); | |
| | read($A$); |
| | $A := A + 50$; |
| | write($A$); |
| | commit; |
| read($B$); | |
| $B := B + 100$; | |
| write($B$); | |
| commit; | |

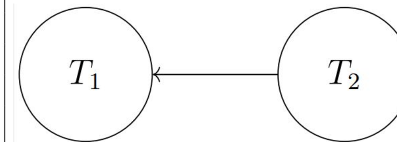


| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| write($A$); | |
| | read($A$); |
| | $A := A + 50$; |
| | write($A$); |
| | commit; |
| read($B$); | |
| $B := B + 100$; | |
| write($B$); | |
| commit; | |



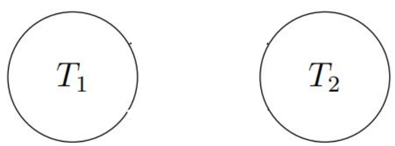



| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| write($A$); | |
| | read($A$); |
| | $A := A + 50$; |
| | write($A$); |
| | commit; |
| read($B$); | |
| $B := B + 100$; | |
| write($B$); | |
| commit; | |



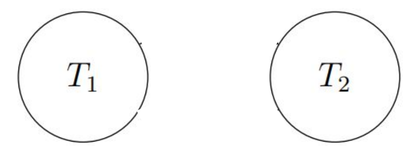



| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| | read($A$); |
| | $A := A + 50$; |
| | write($A$); |
| | commit; |
| write($A$); | |
| read($B$); | |
| $B := B + 100$; | |
| write($B$); | |
| commit; | |

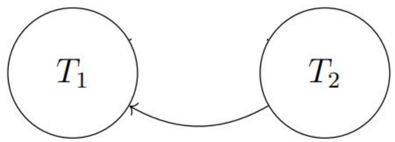





| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| | read($A$); |
| | $A := A + 50$; |
| | write($A$); |
| | commit; |
| write($A$); | |
| read($B$); | |
| $B := B + 100$; | |
| write($B$); | |
| commit; | |

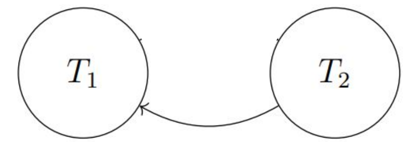


| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| | read($A$); |
| | $A := A + 50$; |
| | write($A$); |
| | commit; |
| write($A$); | |
| read($B$); | |
| $B := B + 100$; | |
| write($B$); | |
| commit; | |



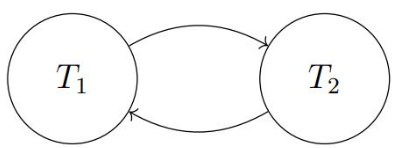



| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| | read($A$); |
| | $A := A + 50$; |
| | write($A$); |
| | commit; |
| write($A$); | |
| read($B$); | |
| $B := B + 100$; | |
| write($B$); | |
| commit; | |



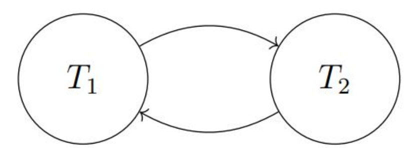



| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| | read($A$); |
| | $A := A + 50$; |
| | write($A$); |
| | commit; |
| write($A$); | |
| read($B$); | |
| $B := B + 100$; | |
| write($B$); | |
| commit; | |






| $T_1$ | $T_2$ |
|---|---|
| read($A$); | |
| $A := A - 100$; | |
| | read($A$); |
| | $A := A + 50$; |
| | write($A$); |
| | commit; |
| write($A$); | |
| read($B$); | |
| $B := B + 100$; | |
| write($B$); | |
| commit; | |




Se **genera un ciclo**, por lo tanto **no es serializable por conflictos**

Existen protocolos para el manejo de la concurrencia

😁

## Optimistas

Asume que los conflictos **son pocos** comunes

😥

## Pesimistas

Asume que los conflictos **son** comunes



😁

## Optimistas

Asume que los conflictos **son pocos** comunes

😥

## Pesimistas

Asume que los conflictos **son** comunes



# Se basan en candados 🔒

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| read(A); | |
| write(A); | |
| **UNLOCK(A);** | |

| Item | Ti | Tipo |
|---|---|---|
| | | |
| | | |
| | | |



# Se basan en candados 🔒

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| read(A); | |
| write(A); | |
| **UNLOCK(A);** | |

| Item | Ti | Tipo |
|---|---|---|
| | | |
| | | |
| | | |

## Se basan en candados

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** read(A); write(A); **UNLOCK(A);** | El candado es otorgado |

| Item | Ti | Tipo |
|---|---|---|
| A | T1 | X |
| | | |
| | | |



## Se basan en candados

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** read(A); write(A); **UNLOCK(A);** | |

| Item | Ti | Tipo |
|---|---|---|
| A | T1 | X |
| | | |
| | | |



## Se basan en candados

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** read(A); write(A); **UNLOCK(A);** | |

| Item | Ti | Tipo |
|---|---|---|
| A | T1 | X |
| | | |
| | | |



## Se basan en candados

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** read(A); write(A); **UNLOCK(A);** | |

| Item | Ti | Tipo |
|---|---|---|
| A | T1 | X |
| | | |
| | | |

# Se basan en candados 🔒

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| read(A); | |
| write(A); | |
| **UNLOCK(A);** | Se libera el candado |

| Item | Ti | Tipo |
|---|---|---|
| | | |
| | | |
| | | |



Hay distintos tipos de candados…

- **Shared lock** ($S - LOCK$). Several transactions can read at the same time, but none can write. This lock can be acquired by multiple transactions at the same time.
- **Exclusive lock** ($X - LOCK$). Only one transaction can both read and write. This lock prevents other transactions acquiring $S - LOCK$ or $X - LOCK$.

| | $S - LOCK$ | $X - LOCK$ |
|---|---|---|
| $S - LOCK$ | ✓ | ✗ |
| $X - LOCK$ | ✗ | ✗ |



Aún con candados, se pueden generar estados inconsistentes

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| read(A); | |
| write(A); | |
| **UNLOCK(A);** | |
| | **X-LOCK(A);** |
| | write(A); |
| | **UNLOCK(A);** |
| **S-LOCK(A);** | |
| read(A); | |
| **UNLOCK(A);** | |



También puede quedar una transacción *starved*, se queda esperando por un recurso

| $T_1$ | $T_2$ | $T_3$ | $T_4$ | ... |
|---|---|---|---|---|
| **S-LOCK(A);** | | | | ... |
| | **X-LOCK(A);** | | | ... |
| | | **S-LOCK(A);** | | ... |
| **UNLOCK(A);** | | | | ... |
| | | | **S-LOCK(A);** | ... |
| | | **UNLOCK(A);** | | ... |

Otro problema son los deadlocks, donde las transacciones no pueden avanzar por el orden en que se piden los recursos

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| | **S-LOCK(B);** |
| | read(B); |
| | **S-LOCK(A);** |
| write(A); | |
| **UNLOCK(B);** | |

| Item | Ti | Tipo |
|---|---|---|
| | | |
| | | |
| | | |



Otro problema son los deadlocks, donde las transacciones no pueden avanzar por el orden en que se piden los recursos

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| | **S-LOCK(B);** |
| | read(B); |
| | **S-LOCK(A);** |
| write(A); | |
| **UNLOCK(B);** | |

| Item | Ti | Tipo |
|---|---|---|
| | | |
| | | |
| | | |



Otro problema son los deadlocks, donde las transacciones no pueden avanzar por el orden en que se piden los recursos

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | Otorgado |
| | **S-LOCK(B);** |
| | read(B); |
| | **S-LOCK(A);** |
| write(A); | |
| **UNLOCK(B);** | |

| Item | Ti | Tipo |
|---|---|---|
| A | T1 | X |
| | | |
| | | |



Otro problema son los deadlocks, donde las transacciones no pueden avanzar por el orden en que se piden los recursos

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| | **S-LOCK(B);** |
| | read(B); |
| | **S-LOCK(A);** |
| write(A); | |
| **UNLOCK(B);** | |

| Item | Ti | Tipo |
|---|---|---|
| A | T1 | X |
| | | |
| | | |

Otro problema son los deadlocks, donde las transacciones no pueden avanzar por el orden en que se piden los recursos

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| | **S-LOCK(B);** |
| | read(B); |
| | **S-LOCK(A);** |
| write(A); | |
| **UNLOCK(B);** | |

Otorgado

| Item | Ti | Tipo |
|---|---|---|
| A | T1 | X |
| B | T2 | S |
| | | |



Otro problema son los deadlocks, donde las transacciones no pueden avanzar por el orden en que se piden los recursos

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| | **S-LOCK(B);** |
| | read(B); |
| | **S-LOCK(A);** |
| write(A); | |
| **UNLOCK(B);** | |

| Item | Ti | Tipo |
|---|---|---|
| A | T1 | X |
| B | T2 | S |
| | | |



Otro problema son los deadlocks, donde las transacciones no pueden avanzar por el orden en que se piden los recursos

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| | **S-LOCK(B);** |
| | read(B); |
| | **S-LOCK(A);** |
| write(A); | |
| **UNLOCK(B);** | |

| Item | Ti | Tipo |
|---|---|---|
| A | T1 | X |
| B | T2 | S |
| | | |



Otro problema son los deadlocks, donde las transacciones no pueden avanzar por el orden en que se piden los recursos

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| | **S-LOCK(B);** |
| | read(B); |
| | **S-LOCK(A);** |
| write(A); | |
| **UNLOCK(B);** | |

| Item | Ti | Tipo |
|---|---|---|
| A | T1 | X |
| B | T2 | S |
| | | |

Otro problema son los deadlocks, donde las transacciones no pueden avanzar por el orden en que se piden los recursos

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| | **S-LOCK(B);** |
| | read(B); |
| | **S-LOCK(A);** |
| write(A); | |
| **UNLOCK(B);** | |

| Item | Ti | Tipo |
|---|---|---|
| A | T1 | X |
| B | T2 | S |
| | | |



En 2PL, se piden los candados en fases

**Fase de crecimiento**
Se piden los candados

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| **X-LOCK(B);** | |
| read(A); | |
| write(A); | |
| | **X-LOCK(A);** |
| **UNLOCK(A);** | |
| | read(A); |
| | write(A); |
| read(B); | |
| write(B); | |
| **ABORT;** | |

**Candados shorts**
Se liberan al momento que ya no se usa

**Fase de contracción**
Se liberan los candados



En strict 2PL, se piden los candados en fases

**Fase de crecimiento**
Se piden los candados

| $T_1$ | $T_2$ |
|---|---|
| **X-LOCK(A);** | |
| **S-LOCK(B);** | |
| | **X-LOCK(A);** |
| read(B); | |
| **UNLOCK(B);** | |
| read(A); | |
| write(A); | |
| **UNLOCK(A);** | |
| **COMMIT;** | |
| | read(A); |
| | write(A); |
| | **COMMIT;** |

**Candados long**
Se liberan al hacer commit o abort

**Fase de contracción**
Se liberan los candados



| | $S-LOCK$ | $X-LOCK$ |
|---|---|---|
| **2PL** | short | short |
| **Strict 2PL** | short | long |
| **Rigorous 2PL** | long | long |

Cuando sucede un deadlock, se puede manejar de varias maneras…

Detección y recuperación

Prevenión



Se puede detectar deadlocks y luego recuperar el sistema a un estado consistente

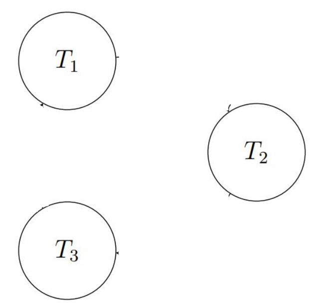


Una manera en que se puede manejar es con wait-for-graphs

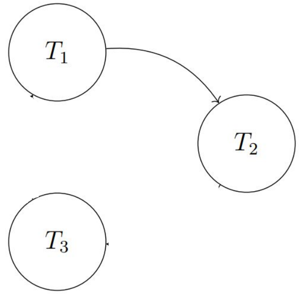
Deadlocks can be detected with *waits-for graphs*. Every transaction is a node. An edge is created from $T_i$ to $T_j$ if $T_i$ is waiting for transaction $T_j$ to release a lock. The edge is removed when $T_j$ releases the data item required by $T_i$. A deadlock exists in the system if, when we check periodically, there is a cycle in the wait-for graph. The deadlock detection algorithm is invoked frequently if many deadlocks occur. Worst case, the algorithm could be invoked every time a lock is not granted immediately.



| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| S-LOCK(A); | | |
| | X-LOCK(B); | |
| | | X-LOCK(C); |
| | S-LOCK(D); | |
| | | S-LOCK(D); |
| S-LOCK(B); | | |
| | S-LOCK(C); | |
| | | X-LOCK(A); |
| ... | ... | ... |

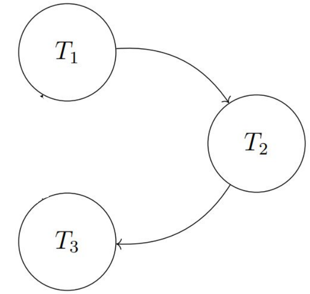


| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| S-LOCK(A); | | |
| | X-LOCK(B); | |
| | | X-LOCK(C); |
| | S-LOCK(D); | |
| | | S-LOCK(D); |
| S-LOCK(B); | | |
| | S-LOCK(C); | |
| | | X-LOCK(A); |
| ... | ... | ... |



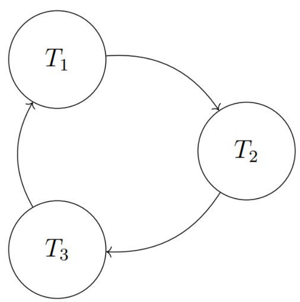



| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| S-LOCK(A); | | |
| | X-LOCK(B); | |
| | | X-LOCK(C); |
| | S-LOCK(D); | |
| | | S-LOCK(D); |
| S-LOCK(B); | | |
| | S-LOCK(C); | |
| | | X-LOCK(A); |
| ... | ... | ... |






| $T_1$ | $T_2$ | $T_3$ |
|---|---|---|
| S-LOCK(A); | | |
| | X-LOCK(B); | |
| | | X-LOCK(C); |
| | S-LOCK(D); | |
| | | S-LOCK(D); |
| S-LOCK(B); | | |
| | S-LOCK(C); | |
| | | X-LOCK(A); |
| ... | ... | ... |






Basado en la detección, se selecciona una víctima para hacerle rollback

En la prevención de deadlocks, se asegura que no se entra en un estado de deadlock



## Hay protocolos que realizan roll back para salir del deadlock

- **Wait-die ("Old waits for young"):** If the requesting transaction has a higher priority than the holding transaction, it waits. Otherwise, it is aborted. The older transaction is allowed to *wait* for a younger transaction. The younger transaction *dies* (aborts) if it requests the lock held by an older transaction.
- **Wound-wait ("Young waits for old"):** If the requesting transaction has a higher priority than the holding transaction, the holding transaction aborts and rolls back (wounds). Otherwise, it waits. The younger transaction is allowed to *wait* for an older transaction. The older transaction *wounds* (abort) the younger transaction holding the lock.



## En *wait-die* T1 espera que T2 libere el candado

| $T_1$ | $T_2$ |
|---|---|
| BEGIN | |
| | BEGIN |
| | **X-LOCK(A);** |
| **X-LOCK(A);** | |

$$T_1 < T_2$$



## En *wound-wait* T2 aborta y T1 obtiene el candado

| $T_1$ | $T_2$ |
|---|---|
| BEGIN | |
| | ~~BEGIN~~ |
| | ~~**X-LOCK(A);**~~ |
| **X-LOCK(A);** | |

$$T_1 < T_2$$

## En *wait-die* T2 pide el candado y como lo tiene T1, T2 muere

| $T_1$ | $T_2$ |
|---|---|
| BEGIN<br>**X-LOCK(A);** | BEGIN<br>**X-LOCK(A);** |

$$T_1 < T_2$$



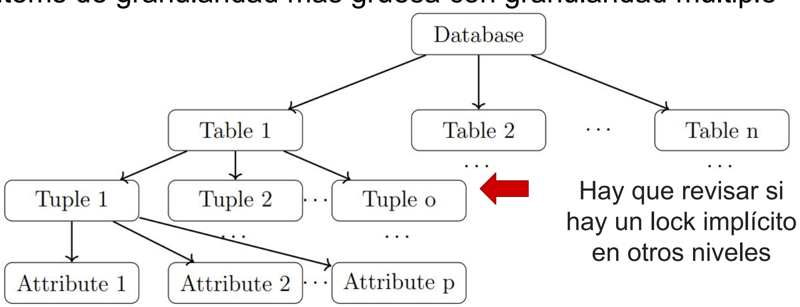
## En *wound-wait* T2 espera que T1 libere el candado

| $T_1$ | $T_2$ |
|---|---|
| BEGIN<br>**X-LOCK(A);** | BEGIN<br>**X-LOCK(A);** |

$$T_1 < T_2$$



## En vez de pedir un candado por elemento, se puede pedir items de granularidad más gruesa con granularidad múltiple




Hay que revisar si hay un lock implícito en otros niveles

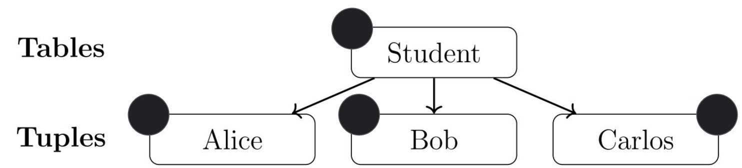


## Por lo tanto, se pueden utilizar nodos de intención para no revisar niveles inferiores

- **Intention-shared (IS) mode:** Indicares that the descendants have explicit shared-mode locks.
- **Intention-exclusive (IX) mode:** Indicates that the descendants have explicit exclusive-mode or shared-mode locks.
- **Shared and intention-exclusive (SIX) mode:** The node is explicitly locked in shared-mode, with the descendants explicitly locked in exclusive-mode.

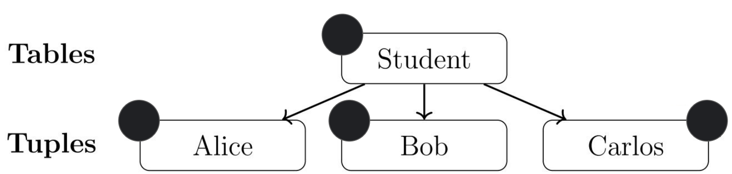
| | $IS$ | $IX$ | $S$ | $SIX$ | $X$ |
|---|---|---|---|---|---|
| $IS$ | ✓ | ✓ | ✓ | ✓ | ✗ |
| $IX$ | ✓ | ✓ | ✗ | ✗ | ✗ |
| $S$ | ✓ | ✗ | ✓ | ✗ | ✗ |
| $SIX$ | ✓ | ✗ | ✗ | ✗ | ✗ |
| $X$ | ✗ | ✗ | ✗ | ✗ | ✗ |





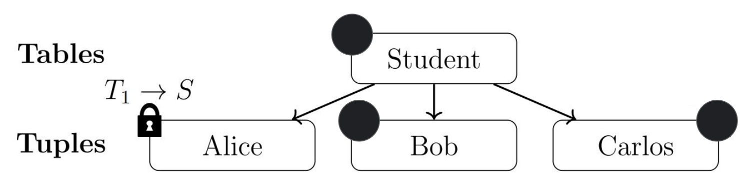



Suppose that transaction $T_1$ wants to read the data for Alice.

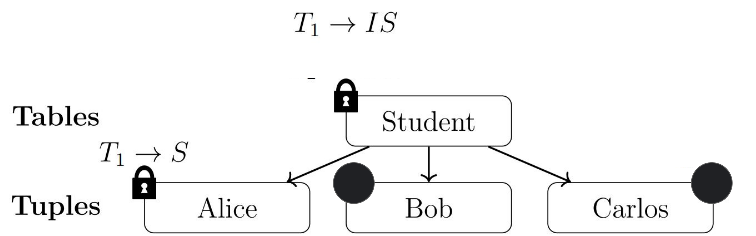





Suppose that transaction $T_1$ wants to read the data for Alice.

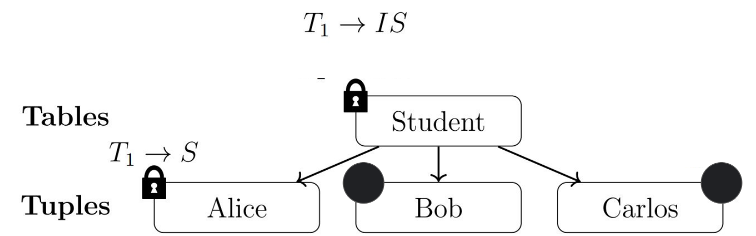


Suppose that transaction $T_1$ wants to read the data for Alice.



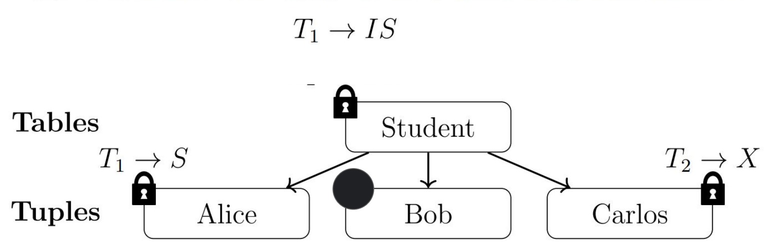



Now, let us assume that transaction $T_2$ wants to update Carlos's record.



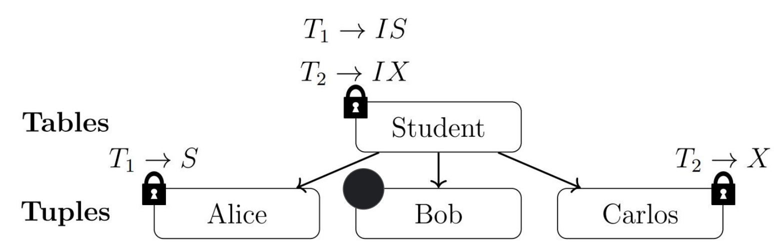



Now, let us assume that transaction $T_2$ wants to update Carlos's record.

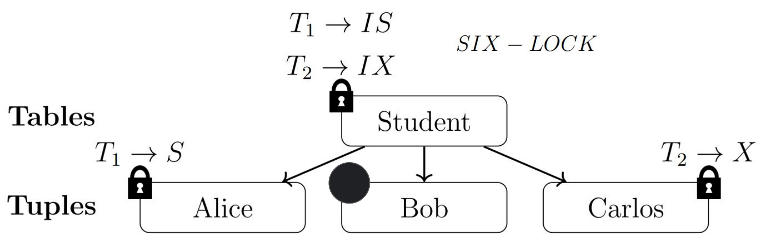





Finally, let's assume that transaction $T_3$ scans wants to scan the student table to update some student records.

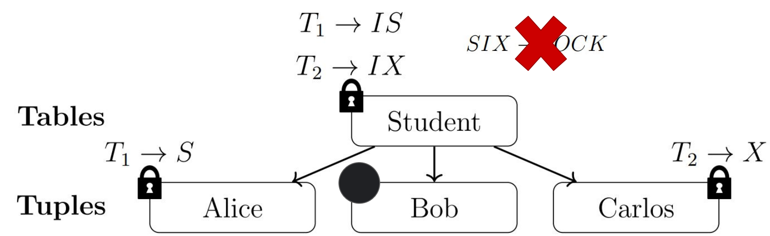


Finally, let's assume that transaction $T_3$ scans wants to scan the student table to update some student records.






Finally, let's assume that transaction $T_3$ scans wants to scan the student table to update some student records.






😁

**Optimistas**

Asume que los conflictos **son pocos** comunes

😥

**Pesimistas**

Asume que los conflictos **son** comunes



## Se puede manejar concurrencia utilizando estampillas de tiempo de las transacciones

- **Read operations.**
  - If $TS(T_i) < W-TS(Q)$, a future transaction has written to data item $Q$ before $T_i$ violating the $TS(T_i) < TS(T_j)$ property. Therefore, $T_i$ is aborted and restarted with a new timestamp value.
  - Else, $TS(T_i) \geq W-TS(Q)$ the order $TS(T_i) < TS(T_j)$ is preserved and the $read(Q)$ instruction is executed. The DBMS also updates $R-TS(Q) = max(R-TS(Q), TS(T_i))$.
- **Write operations.**
  - If $TS(T_i) < R-TS(Q)$ or $TS(T_i) < W-TS(Q)$, a future transaction has read or written to data item $Q$ before $T_i$ violating the $TS(T_i) < TS(T_j)$ property. Therefore, $T_i$ is aborted and restarted with a new timestamp value.
  - Else, $TS(T_i) \geq R-TS(Q)$ and $TS(T_i) \geq W-TS(Q)$ the order of execution is ensured and the $write(Q)$ instruction is executed. The DBMS also updates $W-TS(Q) = TS(T_i)$.

| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

$TS(T_1) = 1$
$TS(T_2) = 2$

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | 0 |
| **B** | 0 | 0 |



| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

$TS(T_1) = 1$
$TS(T_2) = 2$

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | 0 |
| **B** | 0 | 0 |



| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

$\boxed{TS(T_1) = 1}$
$TS(T_2) = 2$

$TS(T_1) \geq W-TS(B) = 1 \geq 0$

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | 0 |
| **B** | 0 | 0 |



| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

$\boxed{TS(T_1) = 1}$
$TS(T_2) = 2$

$R-TS(B) = max(R-TS(B), TS(T_1)) = max(0, 1)$

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | 0 |
| **B** | 0 | 1 |

| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

| | **W-TS** | **R-TS** |
|---|---|---|
| **A** | 0 | 0 |
| **B** | 0 | 1 |

$TS(T_1) = 1$

$TS(T_2) = 2$



$TS(T_2) \geq W-TS(B) = 2 \geq 0$

| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

| | **W-TS** | **R-TS** |
|---|---|---|
| **A** | 0 | 0 |
| **B** | 0 | 1 |

$TS(T_1) = 1$

$\boxed{TS(T_2) = 2}$



$R-TS(B) = max(R-TS(B), TS(T_2)) = max(1, 2)$

| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

| | **W-TS** | **R-TS** |
|---|---|---|
| **A** | 0 | 0 |
| **B** | 0 | 2 |

$TS(T_1) = 1$

$\boxed{TS(T_2) = 2}$



| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

| | **W-TS** | **R-TS** |
|---|---|---|
| **A** | 0 | 0 |
| **B** | 0 | 2 |

$TS(T_1) = 1$

$TS(T_2) = 2$

| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

$TS(T_1) = 1$

$\boxed{TS(T_2) = 2}$

$TS(T_2) \geq W-TS(B) = 2 \geq 0$

$R-TS(B) = 2 \geq 2$

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | 0 |
| **B** | 0 | 2 |



| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

$TS(T_1) = 1$

$\boxed{TS(T_2) = 2}$

$W-TS(B) = TS(T_2) = 2$

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | 0 |
| **B** | 2 | 2 |



| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

$TS(T_1) = 1$

$TS(T_2) = 2$

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | 0 |
| **B** | 2 | 2 |



| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

$\boxed{TS(T_1) = 1}$

$TS(T_2) = 2$

$TS(T_1) \geq W-TS(A) = 1 \geq 0$

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | 0 |
| **B** | 2 | 2 |

| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

$R-TS(A) = max(R-TS(A), TS(T_1)) = max(0,1)$

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | 1 |
| **B** | 2 | 2 |

$\boxed{TS(T_1) = 1}$

$TS(T_2) = 2$



| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | 1 |
| **B** | 2 | 2 |

$TS(T_1) = 1$

$TS(T_2) = 2$



| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

$TS(T_2) \geq W-TS(A) = 2 \geq 0$

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | 1 |
| **B** | 2 | 2 |

$TS(T_1) = 1$

$\boxed{TS(T_2) = 2}$



| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

$R-TS(A) = max(R-TS(A), TS(T_2)) = max(1,2)$

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | 2 |
| **B** | 2 | 2 |

$TS(T_1) = 1$

$\boxed{TS(T_2) = 2}$

| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | 2 |
| **B** | 2 | 2 |

$TS(T_1) = 1$

$TS(T_2) = 2$



$TS(T_1) \geq W-TS(A) = 1 \geq 0$

| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

| | W-TS | R-TS |
|---|---|---|
| **A** | $\boxed{0}$ | 2 |
| **B** | 2 | 2 |

$\boxed{TS(T_1) = 1}$

$TS(T_2) = 2$



$R-TS(A) = max(R-TS(A), TS(T_1)) = max(2,1)$

| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | $\boxed{2}$ |
| **B** | 2 | 2 |

$\boxed{TS(T_1) = 1}$

$TS(T_2) = 2$



| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

| | W-TS | R-TS |
|---|---|---|
| **A** | 0 | 2 |
| **B** | 2 | 2 |

$TS(T_1) = 1$

$TS(T_2) = 2$

| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

$TS(T_1) = 1$

$\boxed{TS(T_2) = 2}$

$TS(T_2) \geq W-TS(A) = 2 \geq 0$

$TS(T_2) \geq R-TS(a) = 2 \geq 2,$

| | W-TS | R-TS |
|---|---|---|
| A | 0 | 2 |
| B | 2 | 2 |



| $T_1$ | $T_2$ |
|---|---|
| read(B); | |
| | read(B); |
| | write(B); |
| read(A); | |
| | read(A); |
| read(A); | |
| | write(A); |

$TS(T_1) = 1$

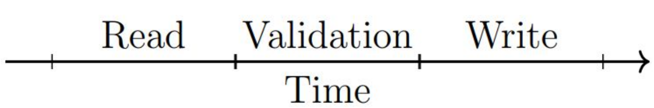
$\boxed{TS(T_2) = 2}$

$W-TS(A) = TS(T_2) = 2.$

| | W-TS | R-TS |
|---|---|---|
| A | 2 | 2 |
| B | 2 | 2 |

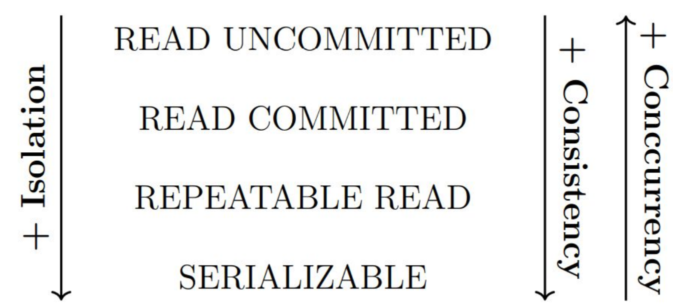


Validación optimista se enfoca en realizar validación solo cuando se hacen escrituras






Los niveles de aislamiento el grado de concurrencia que se requiere



Los problemas que pueden ocurrir dependiendo el nivel de aislamiento son…

| | Lost update | Dirty read | Unrepeatable read | Phantom read |
|---|---|---|---|---|
| READ UNCOMMITTED | ✗ | ? | ? | ? |
| READ COMMITTED | ✗ | ✗ | ? | ? |
| REPEATABLE READ | ✗ | ✗ | ✗ | ? |
| SERIALIZABLE | ✗ | ✗ | ✗ | ✗ |



| | $S-LOCK$ | | $X-LOCK$ |
|---|---|---|---|
| | data-item | condition | |
| READ UNCOMMITTED | None | None | Long |
| READ COMMITTED | Short | Short | Long |
| REPEATABLE READ | Long | Short | Long |
| SERIALIZABLE | Long | Long | Long |



SQL Copy

```sql
USE AdventureWorks2012;
GO
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
GO
BEGIN TRANSACTION;
GO
SELECT *
    FROM HumanResources.EmployeePayHistory;
GO
SELECT *
    FROM HumanResources.Department;
GO
COMMIT TRANSACTION;
GO
```



## Referencias


- R. Elmasri and S. Navathe, Fundamentals of database systems, 7th ed. Pearson, 2016, chapters 20 and 21.
- A. Crotty and M. Li. Lectures #15 to #18. [Online]. Available: https://15445.courses.cs.cmu. edu/fall2021/schedule.html
- A. Silberschatz, H. F. Korth, and S. Sudarshan, Database System Concepts, 7th ed. New York, NY: McGraw-Hill, 2020, chapters 17 and 18.
- H. Berenson, “A Critique of ANSI SQL Isolation Levels,” p. 10.
- H. T. Kung and J. T. Robinson, “On Optimistic Methods for Concurrency Control,” ACM Transactions on Database Systems, vol. 6, no. 2, p. 14.
- Microsoft. (2022). SET TRANSACTION ISOLATION LEVEL (Transact-SQL). Recuperado de: https://docs.microsoft.com/en-us/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-ver16