Transaction processing

DR. MIGUEL ÁNGEL OROS HERNÁNDEZ

Agenda

Transaction processing

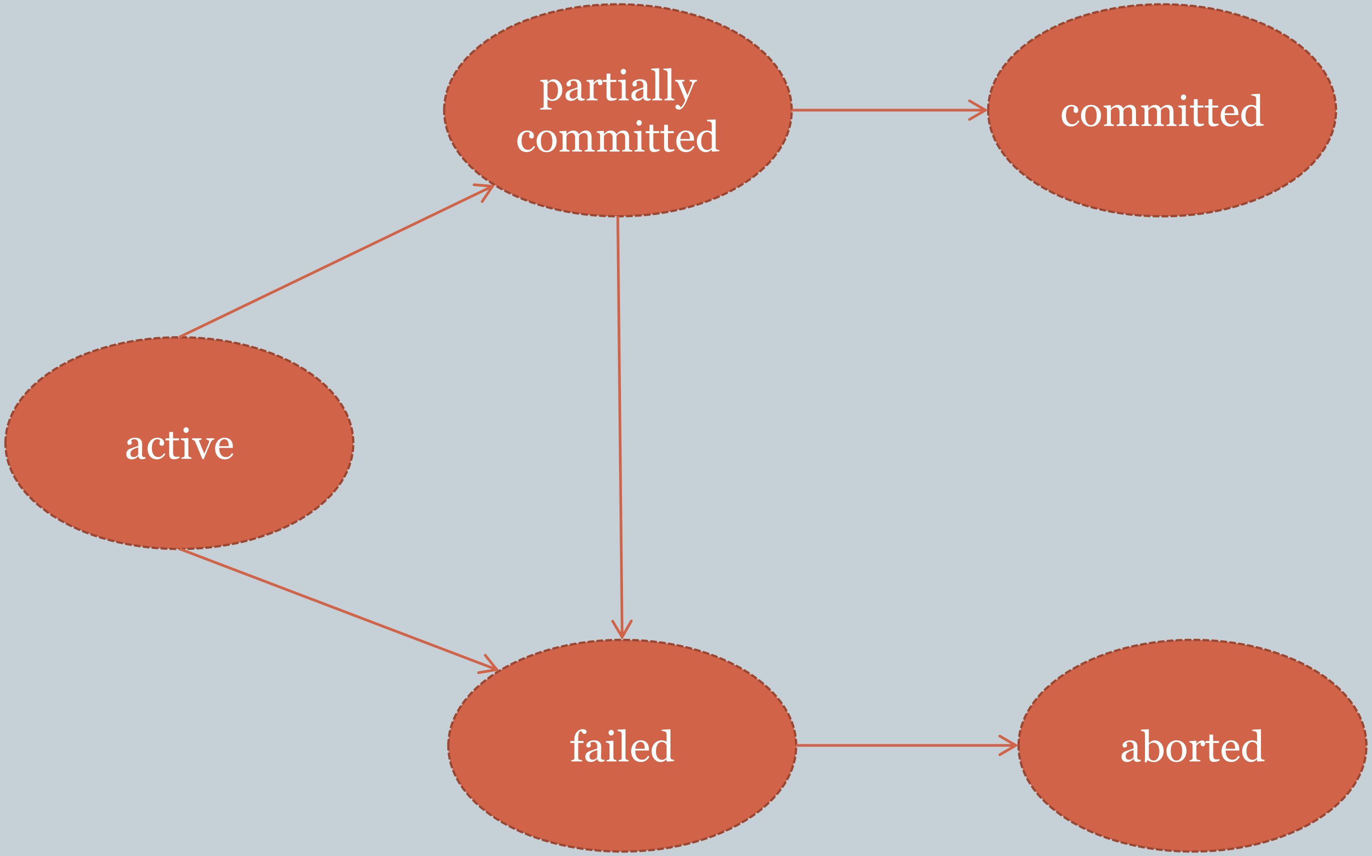
- Transaction properties
- Recovery and serialization algorithms
- Transaction support in SQL
- Concurrency control
- Two phase blocking
- Timestamps
- Deferred and immediate updates

Transacciones

- Conjunto de tareas que se ejecutan como una sola unidad
- Propiedades ACID
- Resultados: éxito o fracaso
- Sentencias
    - Begin transaction
    - Rollback transaction
    - Commit transaction

Transacciones

diagrama de estados



Agenda

Administración de transacciones

1. Definición de transacción
2. Propiedades de las transacciones
3. Tipos de transacciones
4. Beneficios de las transacciones
5. Algunos puntos sobre el procesamiento de transacciones
6. Arquitectura revisada

Definición de transacción

Database consistency

- A database is in a *consistency state* if it obeys all of the consistency (integrity) constraints defined over it
- State changes occur due to modifications, insertions, and deletions
- Objective: ensure that the database never enters an inconsistent state

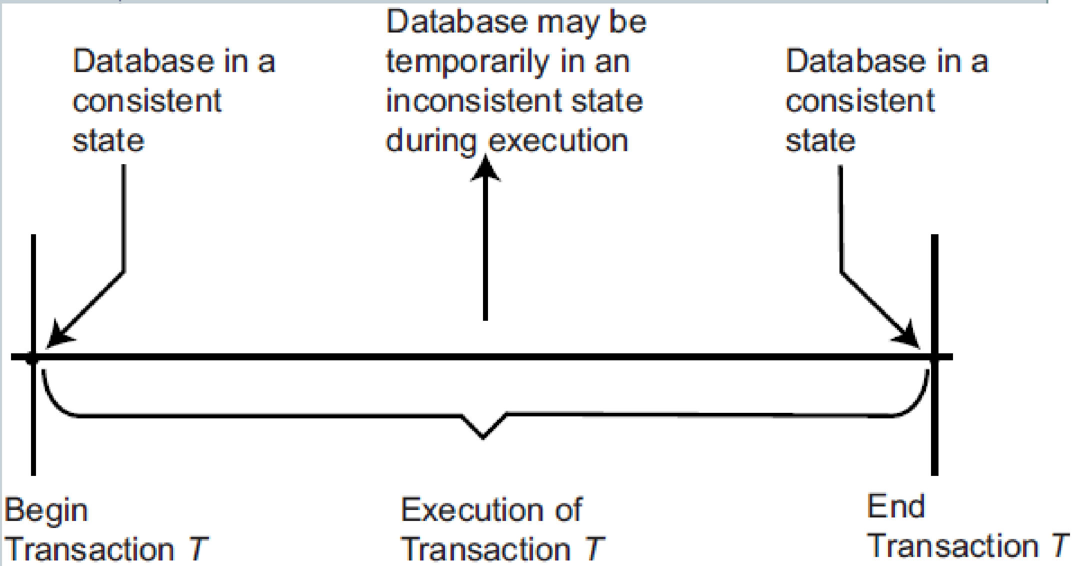
Transaction consistency

- Refers to the actions of concurrent transactions
- Objective: ensure the database remains in a consistent state even if there are a number of user requests that are concurrently accessing (reading or updating) the databases

Definición de transacción

A *transaction* is a collection of actions that make consistent transformations of system states while preserving system consistency

- *Concurrency transparency*
- *Failure transparency*



Definición de transacción

transaction example – a simple SQL Query

```
begin transaction BUDGET_UPDATE
begin
   UPDATE PROJ
   SET BUDGET = BUDGET*1.1
   WHERE PNAME = "ECOMMERCE"
end
```

Definición de transacción

example database: airline reservation and transaction example

```
FLIGHT(FNO, DATE, SRC, DEST, STSOLD, CAP)
CUST(CNAME, ADDR, BAL)
FC(FNO, DATE, CNAME, SPECIAL)
```

```
begin transaction Reservation
begin
  input(fligh_no, date, customer_name)
  UPDATE FLIGHT
  SET STSOLD = STSOLD + 1
  WHERE FNO = flight_no AND DATE = date
  INSERT FC(FNO, DATE, CNAME, SPECIAL)
  VALUES (flight_no, date, customer_name, null)
end {Reservation}
```

Definición de transacción

example of transaction – reads and writes

```
begin transaction Reservation
begin
  input(flight_no, date, customer_name)
  SELECT STSOLD as temp1, CAP INTO temp2
  FROM FLIGHT
  WHERE FNO = flight_no AND DATE = date
  if temp1 = temp2 then
    Abort
  else
    UPDATE FLIGHT SET STSOLD = STSOLD + 1
    WHERE FNO = flight_no AND DATE = date

    INSERT FC(FNO, DATE, CNAME, SPECIAL)
    VALUES (flight_no, date, customer_name, null)
    Commit
  endif
end { Reservation }
```

Definición de transacción

termination of transactions

```
begin transaction Reservation
begin
  input(flight_no, date, customer_name)
  temp ← Read(flight_no(date),stsold)
  if temp = flight(date).cap then
    output("no free seats")
    Abort
  else
    Write(flight(date).stsold, temp + 1)
    Write(flight(date).cname, customer_name)
    Write(flight(date).special, null)
    Commit
    output("reservation completed")
  endif
end { Reservation }
```

Definición de transacción

characterization

- Read set (RS)

  The set of data items that are read by a transaction

- Write set (WS)

  The set of data items that whose values are changed by this transaction

- Base set (BS)

$$RS \cup WS$$

Definición de transacción

characterization: example

$$RS[Reservation] = \{FLIGHT.STSOLD, FLIGHT.CAP\}$$

$$WS[Reservation] = \left\{\begin{matrix} FLIGHT.STSOLD, FLIGHT.FNO, \\ FC.DATE, \\ FC.CNAME, FC.SPECIAL \end{matrix}\right\}$$

$$BS[Reservation] = \left\{\begin{matrix} FLIGHT.STSOLD, FLIGHT.CAP, \\ FC.FNO, FC.DATE, FC.CNAME, \\ FC.SPECIAL \end{matrix}\right\}$$

Definición de transacción

formalization

- $O_{ij}(x)$: operation O_j of transaction T_i that operates on a database entity x

  where

  - $O_{ij} \in \{read, write\}$
  - O_j is atomic (i.e. each is executed as an indivisible unit)

- OS_i: the set of all operations in T_i

$$OS_i = \bigcup_j O_{ij}$$

- N_i: the termination condition for T_i, where $N_i \in \{abort, commit\}$

Definición de transacción

formalization

Transaction T_i is a partial order $T_i = \{\Sigma_i, \prec_i\}$ where

❶ $\Sigma_i = OS_i \cup \{N_i\}$

❷ For any two operations O_{ij} , $O_{ik} \in OS_i$, if $O_{ij} = R(x)$ and $O_{ik} = W(x)$ for any data item x, then either $O_{ij} \prec_i O_{ik}$ or $O_{ik} \prec_i O_{ij}$

❸ $O_{ij} \in OS_i$, $O_{ij} \prec_i N_i$

Definición de transacción

formalization: operations in conflict

Two operations, O_i and O_j, are said to be *in conflict* if $O_i = Write$ or $O_j = Write$ (i.e., at least one of them is a $Write$ and they access the same data item)

Definición de transacción

formalization: example 1

Consider a transaction *T*:

Read(x)

Read(y)

x $\leftarrow$ x + y

Write(x)

Commit

Then

Σ = {R(x), R(y), W(x), C}

$\prec$ = {(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)}

where

(O_i, O_j) as an element of the $\prec$ relation indicates that $O_i \prec O_j$

Definición de transacción

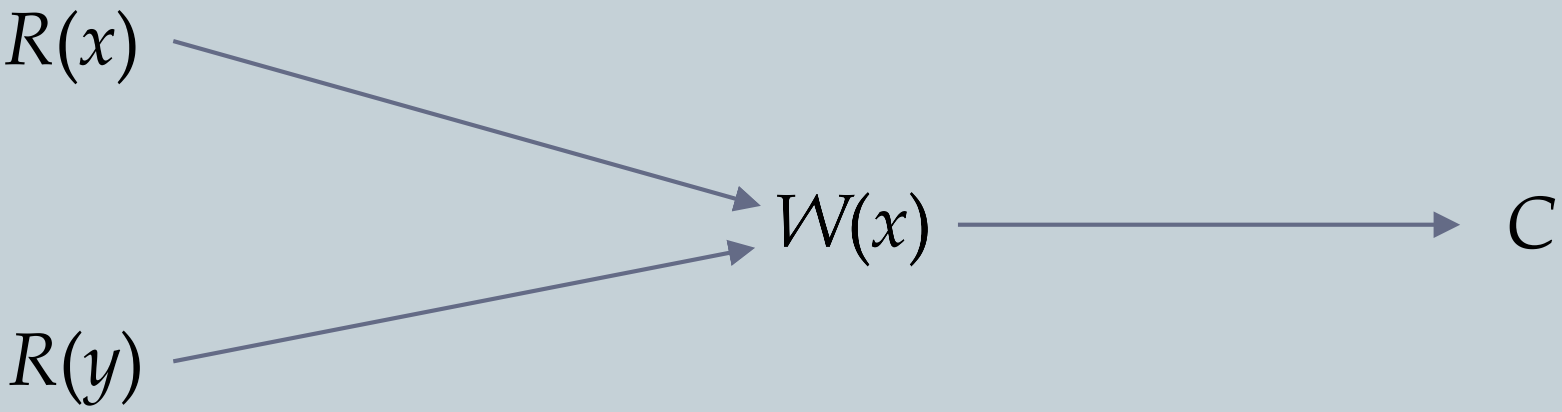
formalization: DAG representation

Assume

$T = \{R(x), R(y), W(x), C\}$

$\prec = \{(R(x), W(x)), (R(y), W(x)), (W(x), C), (R(x), C), (R(y), C)\}$

Direct Acyclic Graph (DAG)



Definición de transacción

formalization: example 2: the reservation transaction

There are two possible termination conditions, depending on the availability of seats

$\Sigma = \{R(STSOLD), R(CAP), A\}$

$\prec = \{(O_1, A), (O_2, A)\}$

$\Sigma = \{R(STSOLD), R(CAP), W(STOLD), W(FNO), W(DATE), W(CNAME)\ W(SPECIAL), C\}$

$\prec = \{(O_1, O_3), (O_2, O_3), (O_1, O_4), (O_1, O_5), (O_1, O_6), (O_1, O_7), (O_2, O_4), (O_2, O_5), (O_2, O_6), (O_2, O_7), (O_1, C), (O_2, C), (O_3, C), (O_4, C), (O_5, C), (O_6, C), (O_7, C)\}$

where

$O_1 = R(STSOLD)$, $O_2 = R(CAP)$, $O_3 = W(STSOLD)$, $O_4 = W(FNO)$,

O =R(DATE), O =W(CNAME), O =W(SPECIAL)

Propiedades de las transacciones

Atomicity

All or nothing

Consistency

No violation of integrity constraints

Isolation

Concurrent changes invisible $\Longrightarrow$ serializable

Durability

Committed update persist

Propiedades de las transacciones

Atomicity

- Either all or none of the transaction's operations are performed
- Atomicity requires that if a transaction is interrupted by a failure, its partial results must be undone
- The activity of preserving the transaction's atomicity in presence of transaction aborts due to input errors, system overloads, or deadlocks is called transaction recovery
- The activity of ensuring atomicity in the presence of system crashes is called crash recovery

Propiedades de las transacciones

Consistency

- Internal consistency
  - A transaction which executes alone against a consistent database leaves it in a consistent state
  - Transactions do not violate database integrity constraints
- Transactions are correct programs

Propiedades de las transacciones

Consistency: consistency degrees

- Degree 0
  - Transaction T does not overwrite dirty data of other transactions
  - Dirty data refers to data values that have been updated by a transaction prior to its commitment
- Degree 1
  - T does not overwrite dirty data of other transactions
  - T does not commit any writes before End Of Transaction (EOT)

Propiedades de las transacciones

Consistency: consistency degrees

- Degree 2
  - T does not overwrite dirty data of other transactions
  - T does not commit any writes before EOT
  - T does not read dirty data from other transactions
- Degree 3
  - T does not overwrite dirty data of other transactions
  - T does not commit any writes before EOT
  - T does not read dirty data from other transactions
  - Other transactions do not dirty any data read by T before T completes

Propiedades de las transacciones

Isolation

- Serializability

  If several transactions are executed concurrently, the results must be the same as if they were executed serially in some order

- Incomplete results
  - An incomplete transaction cannot reveal its results to other transactions before its commitment
  - Necessary to avoid cascading aborts

Propiedades de las transacciones

Isolation: example

Consider the following two transactions:

T_1:	Read(x)	T_2:	Read(x)
	$x \leftarrow x+1$		$x \leftarrow x+1$
	Write(x)		Write(x)
	Commit		Commit

Possible execution sequences:

T_1:	Read(x)	T_1:	Read(x)
T_1:	$x \leftarrow x+1$	T_1:	$x \leftarrow x+1$
T_1:	Write(x)	T_2:	Read(x)
T_1:	Commit	T_1:	Write(x)
T_2:	Read(x)	T_2:	$x \leftarrow x+1$
T_2:	$x \leftarrow x+1$	T_2:	Write(x)
T_2:	Write(x)	T_1:	Commit
T_2:	Commit	T_2:	Commit

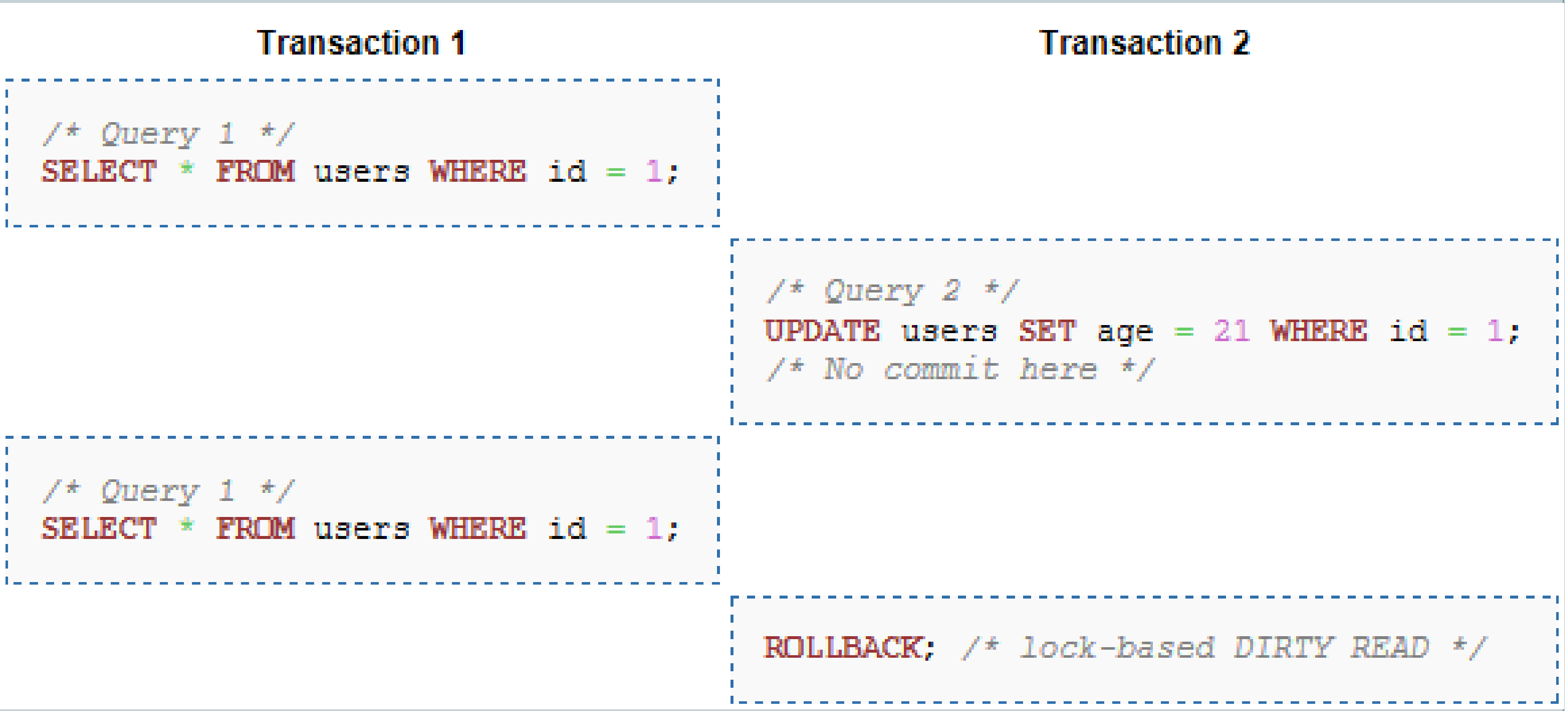
Propiedades de las transacciones

Isolation: SQL-92 Isolation Levels

- Dirty read
  - T_1 modifies x which is then read by T_2 before T_1 terminates
  - T_1 aborts, T_2 has read value which never exists in the database
- Non-repeatable (fuzzy) read
  - T_1 reads x
  - T_2 then modifies or deletes x and commits
  - T_1 tries to read x again but reads a different value or can't find it
- Phantom
  - T_1 searches the database according to a predicate while T_2 inserts new tuples that satisfy the predicate

Propiedades de las transacciones

Isolation: SQL-92 Isolation Levels: Dirty Read



Transaction 1

Transaction 2

Transaction 1:

```sql
/* Query 1 */
SELECT * FROM users WHERE id = 1;
```

Transaction 2:

```sql
/* Query 2 */
UPDATE users SET age = 21 WHERE id = 1;
/* No commit here */
```

Transaction 1:

```sql
/* Query 1 */
SELECT * FROM users WHERE id = 1;
```

Transaction 2:

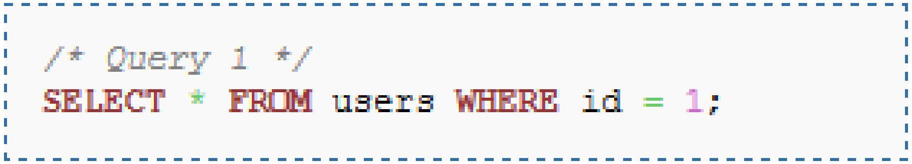
```sql
ROLLBACK; /* lock-based DIRTY READ */
```

Propiedades de las transacciones

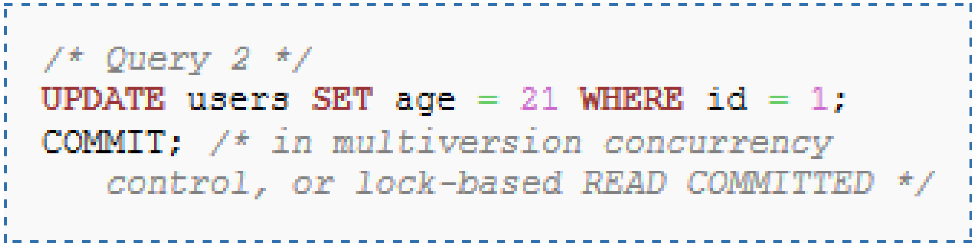
Isolation: SQL-92 Isolation Levels: Non-repeatable (fuzzy) read

Transaction 1

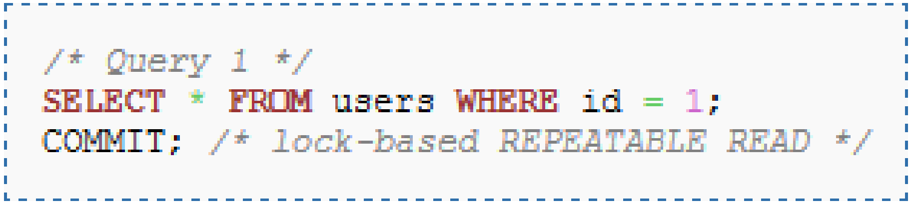
Transaction 2



```sql
/* Query 1 */
SELECT * FROM users WHERE id = 1;
```



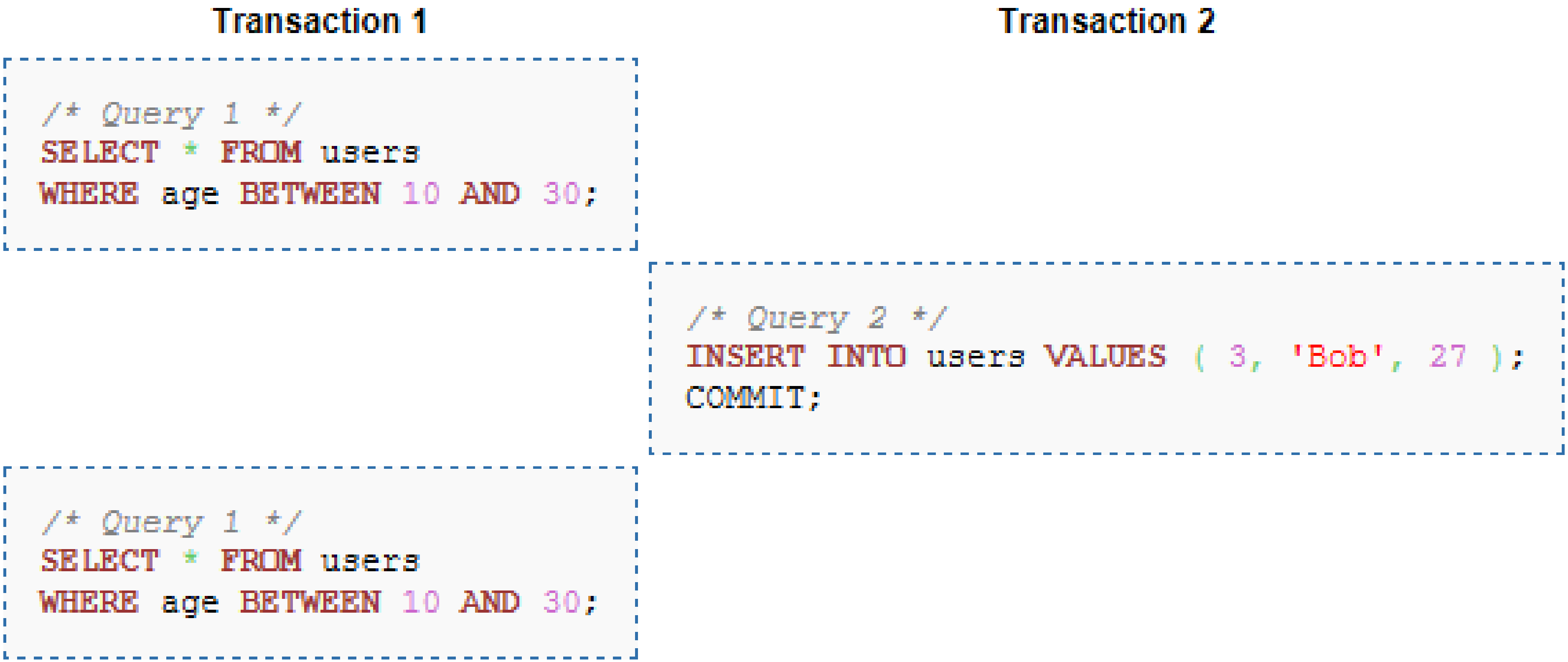
```sql
/* Query 2 */
UPDATE users SET age = 21 WHERE id = 1;
COMMIT; /* in multiversion concurrency
   control, or lock-based READ COMMITTED */
```



```sql
/* Query 1 */
SELECT * FROM users WHERE id = 1;
COMMIT; /* lock-based REPEATABLE READ */
```

Propiedades de las transacciones

Isolation: SQL-92 Isolation Levels: Phantom Read



Transaction 1

```
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
```

Transaction 2

```
/* Query 2 */
INSERT INTO users VALUES ( 3, 'Bob', 27 );
COMMIT;
```

Transaction 1

```
/* Query 1 */
SELECT * FROM users
WHERE age BETWEEN 10 AND 30;
```

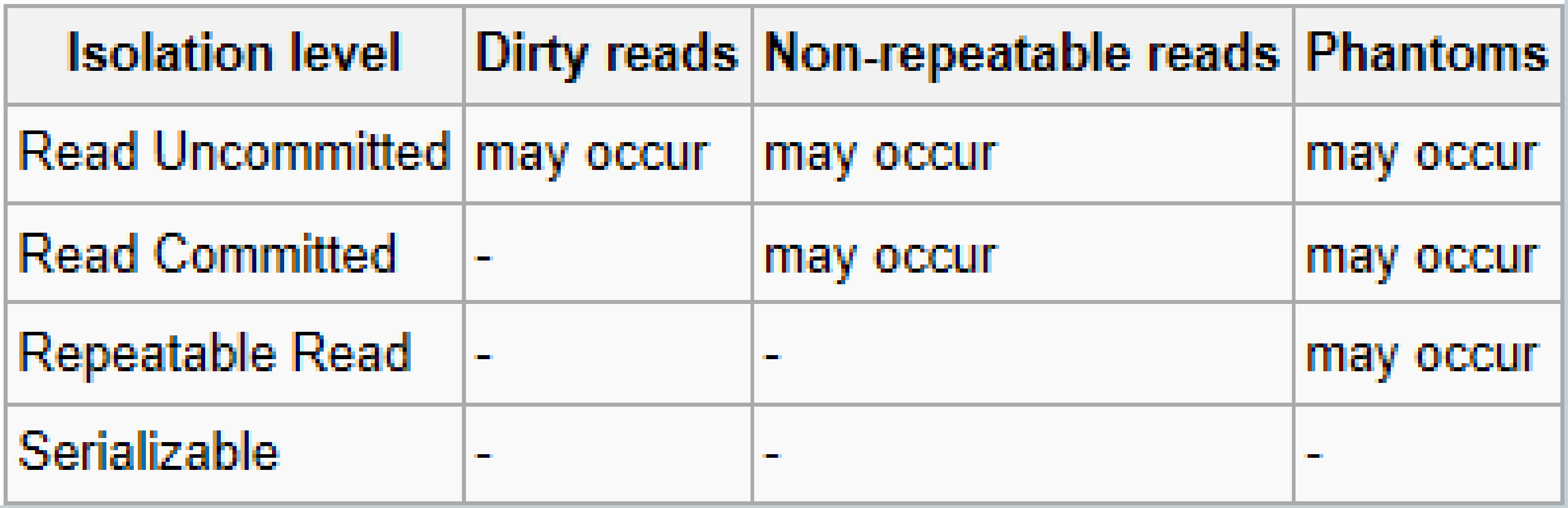
Propiedades de las transacciones

Isolation: SQL-92 Isolation Levels

- Read Uncommitted
  - For transactions operating at this level, all three phenomena are possible
- Read Committed
  - Fuzzy reads and phantoms are possible, but dirty reads are not
- Repeatable Read
  - Only phantoms possible
- Anomaly Serializable
  - None of the phenomena are possible

Propiedades de las transacciones

Isolation: SQL-92 Isolation Levels:Isolation Levels vs Read Phenomena

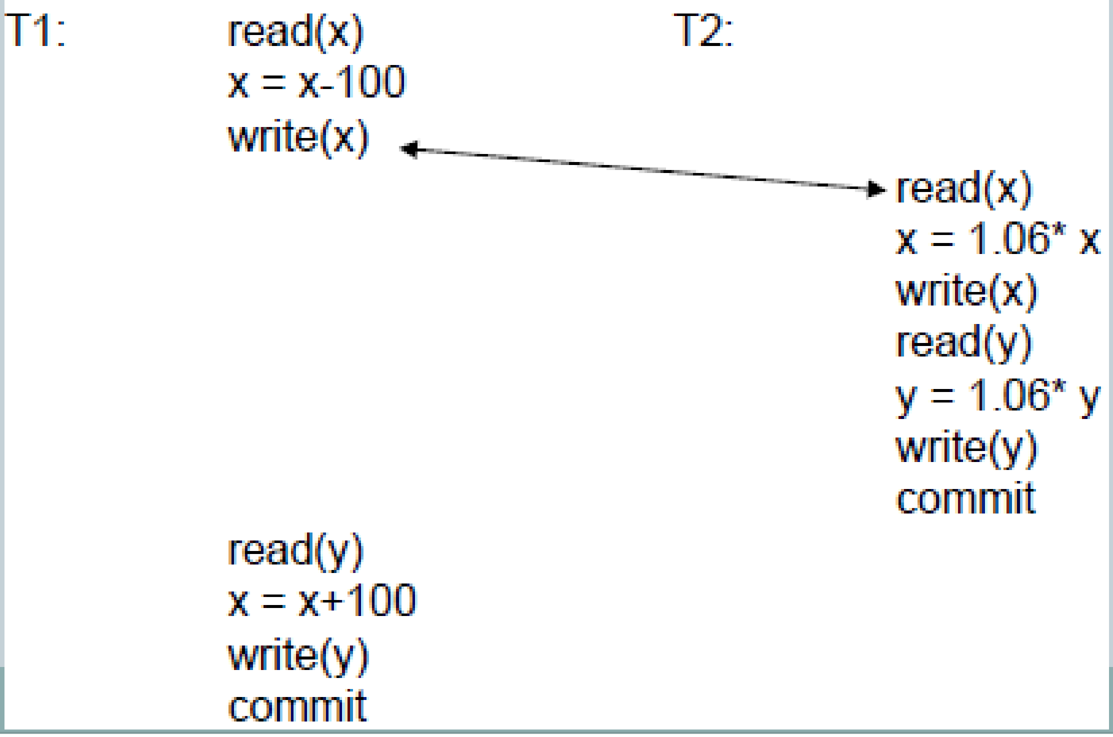


Isolation level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur
Read Committed	-	may occur	may occur
Repeatable Read	-	-	may occur
Serializable	-	-	-

Propiedades de las transacciones

Dirty Read (Lost Update) Problem

- Reading uncommitted data – write-read conflicts
- Example
  - T1 transfers $100 from one account to another
  - T2 adds 6% to each account



T1:	T2:
read(x)	
x = x-100	
write(x)	
	read(x)
	x = 1.06* x
	write(x)
	read(y)
	y = 1.06* y
	write(y)
	commit
read(y)	
x = x+100	
write(y)	
commit	

Propiedades de las transacciones

Dirty Read (Lost Update) Problem

- Suppose Account X had 200 and Account Y had 100
- If T1 runs entirely before T2
  - Account X transfers 100 to Account Y
  - X=100 and Y=200
  - Then T2 adds 6% → X=106 and Y=212
  - X+Y=318
- If T2 runs entirely before T1
  - T2 adds 6 % → X=212 and Y=106
  - Account X transfers 100 to Account Y
  - X=112 and Y=206
  - X+Y=318

Propiedades de las transacciones

Dirty Read (Lost Update) Problem

- In our scenario
  - After the first T1 operation, X=200, Y = 100
  - T2 adds 6% → X=106, Y=106
  - Then, T1 adds the 100 back into Y, X= 106, Y=206
  - X+Y = 312
- Transaction T1 loses money!

Propiedades de las transacciones

Non-repeatable (Fuzzy) Read Problem

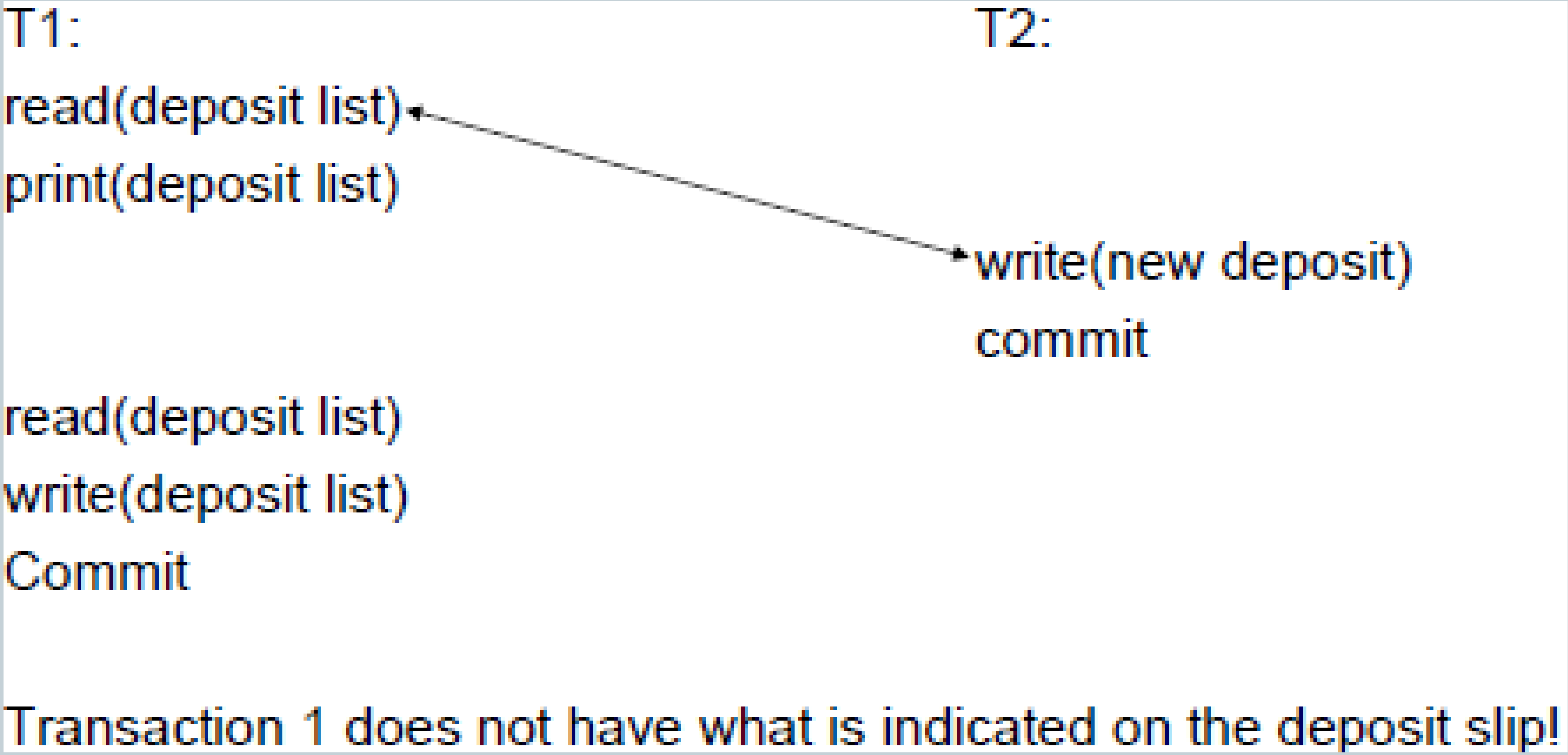
Read-Write Conflicts

T1:	T2:
read(balance) ($**100**)	
balance=balance-100 ($**0**)	
	read(balance) ($**100**)
	balance=balance-50 ($**50**)
	write(balance) ($**50**)
if (balance <= 50)	
read(balance) ($**50**)	
write(balance) ($**50**)	
commit	
	abort

Transaction 1 loses money!

Propiedades de las transacciones

Phantom Reads: Read-Write conflicts



T1:

read(deposit list)

print(deposit list)

T2:

write(new deposit)

commit

read(deposit list)

write(deposit list)

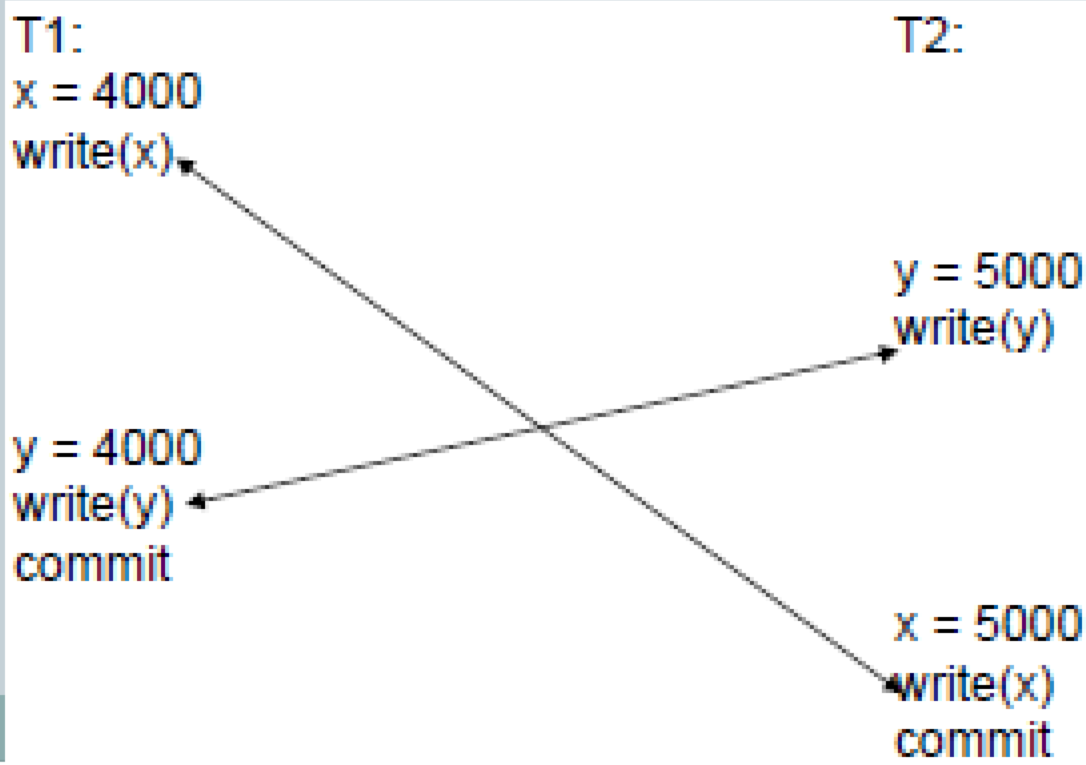
Commit

Transaction 1 does not have what is indicated on the deposit slip!

Propiedades de las transacciones

Overwriting Uncommited Data

- Employees x and y must maintain a consistent salary
- T1 sets both salaries to 4000/month and T2 sets both salaries to 5000/month
- Neither transaction reads their current salaries



T1:	T2:
x = 4000	
write(x)	
	y = 5000
	write(y)
y = 4000	
write(y)	
commit	
	x = 5000
	write(x)
	commit

Propiedades de las transacciones

Durability

- Once a transaction commits, the system must guarantee that the results of its operations will never be lost, in spite of subsequent failures
- In other words, once transaction commits, it is permanent
- Transaction will survive subsequent failures
- Database recovery

Transaction architecture for Distributed DBMS

Need to add a Transaction Manager (TM) and a Scheduler (SC)

- TM coordinates transactions for all applications
- SC implements specific concurrency algorithm for synchronous access to databases
- Need local recovery managers to rollback transactions

Transaction Managers

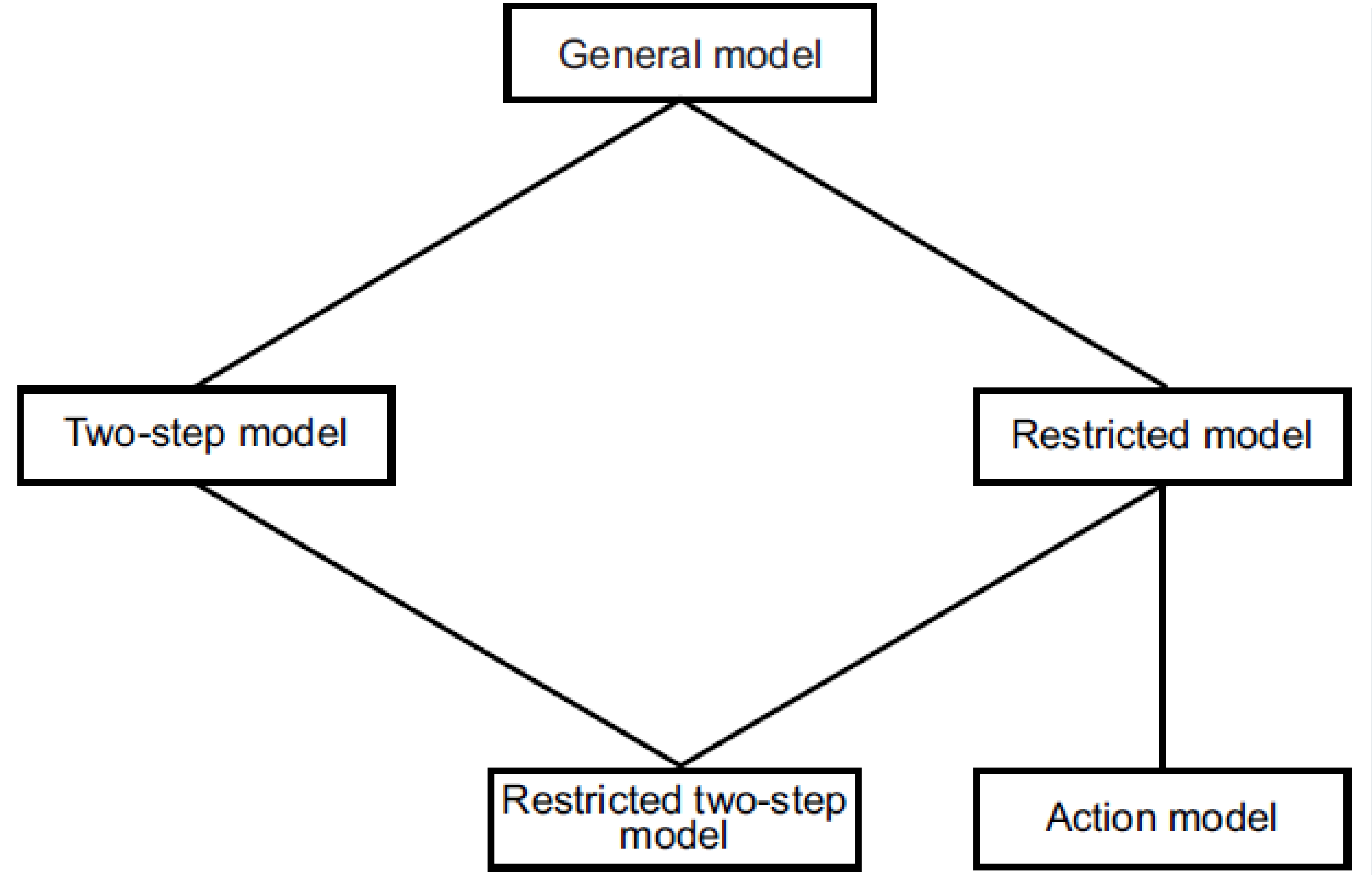
five commands

- Begin
  - Set up new transaction
  - Keep information useful in case a rollback occurs
- Read
  - If the value is local, read it form the local site
  - If not, select one site and read it from there
- Write
  - Write value to all sites that stored the value
- Commit
  - Coordinates all sites to inform them that the write from a transaction is permanent
- Abort
  - Coordinates all sites to inform them that all writes of a transaction must no be permantently recorded

Tipos de transacciones

- Application areas
  - *Non-distributed vs. distributed*
  - *Compensating transactions*
  - *Heterogeneous transactions*
- Timing
  - *On-line (short-life) vs batch (long-life)*
- Organization of read and write actions
  - *Two-step*
  - *Restricted*
  - *Action model*
- Structure
  - *Flat (or simple) transactions*
  - *Nested transactions*
  - *Workflows*

Tipos de transacciones



General model

Two-step model

Restricted model

Restricted two-step model

Action model

Tipos de transacciones

Examples

- General

$$T_1 : \{R(x), R(y), W(y), R(z), W(x), W(z), W(w), C\}$$

- Two-step

$$T_2 : \{R(x), R(y), R(z), W(x), W(z), W(y), W(w), C\}$$

- Restricted

$$T_3 : \{R(x), R(y), W(y), R(z), W(x), W(z), R(w), W(w), C\}$$

Tipos de transacciones

Examples

- Two-step restricted

$$T_4 : \{R(x), R(y), R(z), R(w), W(x), W(z), W(y), W(w), C\}$$

- Action

$$T_5 : \{[R(x), W(x)], [R(y), W(y)], [R(z), W(z)], [R(w), W(w)], C\}$$

Tipos de transacciones
Transaction structure

- Flat transaction
  - Consists of a sequence of primitive operations embraced between a **begin** and **end** markers

```
begin_transaction Reservation
   …
end.
```

- Nested transaction
  - The operations of a transaction may themselves be transactions

```
begin_transaction Reservation
   …
   begin_transaction Airline
         …
   end. {Airline}
   begin_transaction Hotel
     …
   end. {Hotel}
end. {Reservation}
```

Tipos de transacciones

Transaction structure: Nested transactions

- Have the same properties as their parents may themselves have other nested transactions
- Introduces concurrency control and recovery concepts to within the transaction
- Types
  - Closed nesting
    - Subtransactions begin after their parents and finish before them
    - Commitment of a subtransaction is conditional upon the commitment of the parent (commitment through the root)
  - Open nesting
    - Subtransactions can execute and commit independently
    - Compensation may be necessary

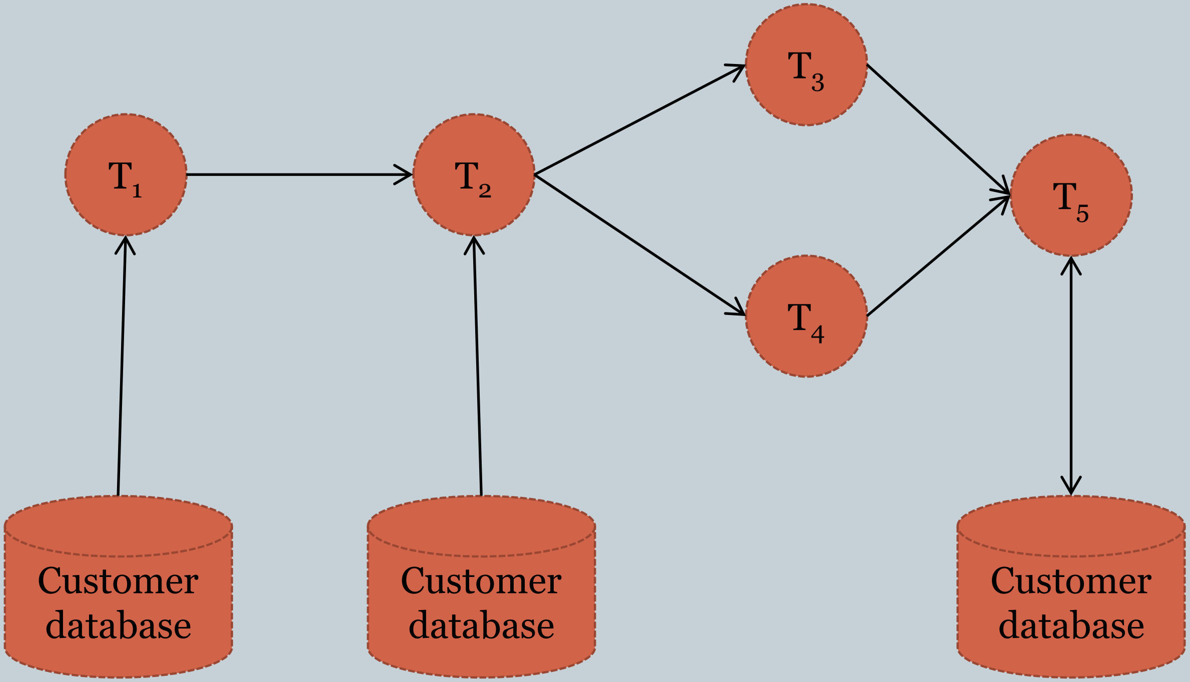
Tipos de transacciones

Transaction structure: Workflows

- "A collection of tasks organized to accomplish some business process."
- Types
  - Human-oriented workflows
    - Involve humans in performing the tasks
    - System support for collaboration and coordination; but no system-wide consistency definition
  - System-oriented workflows
    - Computation-intensive & specialized tasks that can be executed by a computer
    - System support for concurrency control and recovery, automatic task execution, notification, etc.
  - Transactional workflows
    - In between the previous two; may involve humans, require access to heterogeneous, autonomous and/or distributed systems, and support selective use of ACID properties

Tipos de transacciones

Transaction structure: Workflows example



T_1: Customer request obtained

T_2: Airline reservation performed

T_3: Hotel reservation perfomed

T_4: Auto reservation performed

T_5: Bill generated

Beneficios de las transacciones

- *Atomic* and *reliable* execution in the presence of failures
- *Correct* execution in the presence of multiple user accesses
- Correct management of *replicas* (if they support it)

Transacciones

ejemplo 1 …

```
USE pubs

DECLARE @intErrorCode INT

BEGIN TRAN
  UPDATE Authors
  SET Phone = '415 354-9866'
  WHERE au_id = '724-80-9391'

  SELECT @intErrorCode = @@ERROR

  IF (@intErrorCode <> 0) GOTO PROBLEM

  UPDATE Publishers
  SET city = 'Córdoba', country = 'México'
  WHERE pub_id = '9999'
```

Transacciones

… ejemplo 1

```
   SELECT @intErrorCode = @@ERROR

   IF (@intErrorCode <> 0) GOTO PROBLEM
COMMIT TRAN

PROBLEM:
   IF (@intErrorCode <> 0)
   BEGIN
      PRINT '¡Ocurrió un error no previsto!'
      ROLLBACK TRAN
   END
```

Transacciones anidadas

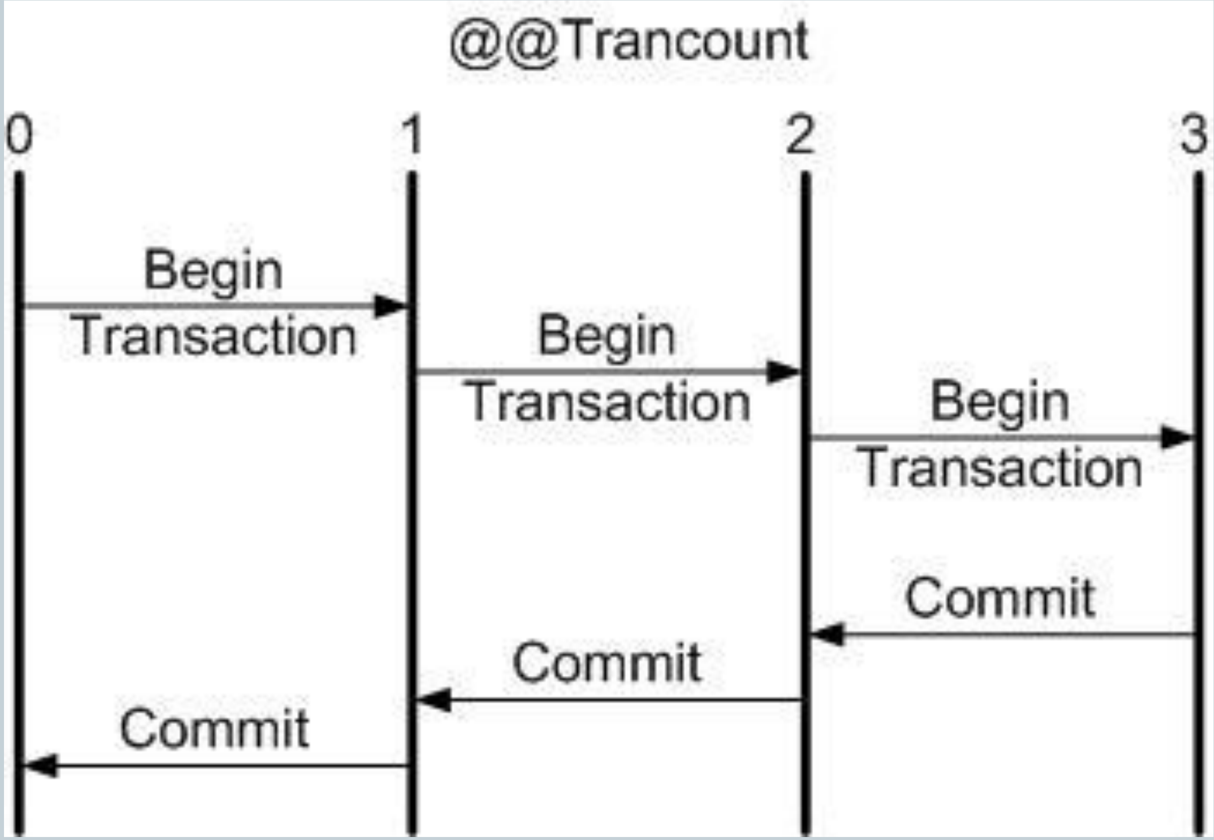
- SQL Server permite anidar transacciones

  Una nueva transacción puede empezar aún si la previa no ha concluido

- `@@TRANCOUNT`
  - Regresa el nivel de anidamiento; así, 0 = no anidación, 1 = un nivel de anidamiento, …
  - Comportamiento no simétrico entre el `commit` y el `rollback`

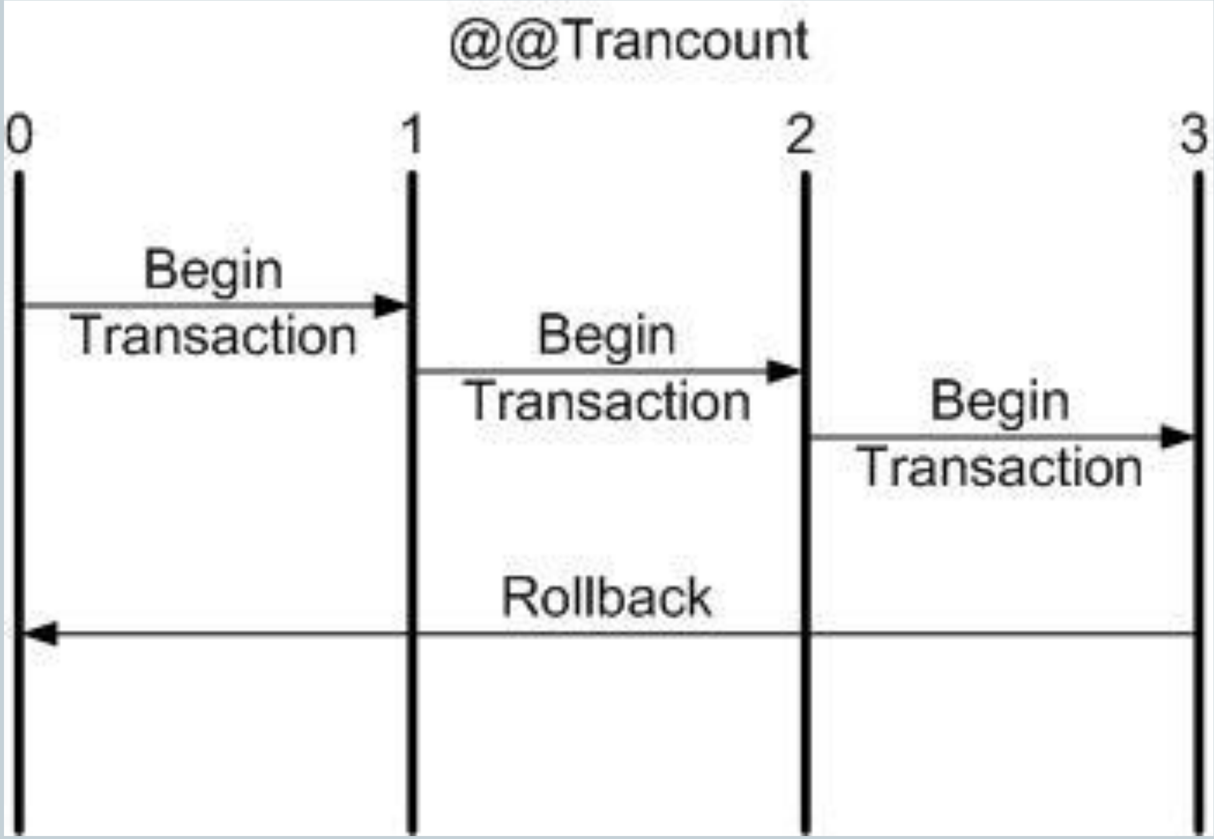
Transacciones anidadas

comportamiento del `commit`



Transacciones anidadas

comportamiento del `rollback`



Transacciones anidadas

ejemplo 2 …

```
USE bancaria

SELECT 'antes del BEGIN TRAN', @@TRANCOUNT
-- valor de @@TRANCOUNT = 0

BEGIN TRAN
  SELECT 'Después BEGIN TRAN', @@TRANCOUNT
  -- valor de @@TRANCOUNT = 1

  DELETE deposit
  BEGIN TRAN nested
    SELECT 'Después BEGIN TRAN nested', @@TRANCOUNT
    -- valor de @@TRANCOUNT = 2
    DELETE borrow
  COMMIT TRAN nested
  -- decrementó de @@TRANCOUNT
```

Transacciones anidadas

… ejemplo 2

```
  SELECT 'Después COMMIT TRAN nested', @@TRANCOUNT
  -- valor de @@TRANCOUNT = 1

ROLLBACK TRAN
SELECT 'Después ROLLBACK TRAN', @@TRANCOUNT
-- valor de @@TRANCOUNT = 0
-- porque ROLLBACK TRAN siempre deshace todas las
-- transacciones y asigna 0 a @@TRANCOUNT

SELECT TOP 5 customer_name FROM deposit
```

Puntos de verificación

savepoints

- Mecanismo para deshacer porciones de transacciones
- Define una ubicación en la cual una transacción puede regresar si una parte de la transacción es cancelada
- Uso de `SAVE TRAN` en SQL Server y no afecta `@@TRANCOUNT`
- Un `rollback` hasta un *savepoint* (no a la transacción) no afecta el valor de `@@TRANCOUNT`

Puntos de verificación

ejemplo …

```
USE bancaria

SELECT 'Before BEGIN TRAN main', @@TRANCOUNT
-- The value of @@TRANCOUNT is 0

BEGIN TRAN main
  SELECT 'After BEGIN TRAN main', @@TRANCOUNT
  -- The value of @@TRANCOUNT is 1

  DELETE deposit

  SAVE TRAN depositos -- Mark a save point

  SELECT 'After SAVE TRAN depositos', @@TRANCOUNT
  -- The value of @@TRANCOUNT is still 1
```

Puntos de verificación

… ejemplo …

```
BEGIN TRAN nested
  SELECT 'After BEGIN TRAN nested', @@TRANCOUNT
  -- The value of @@TRANCOUNT is 2

  DELETE borrow

  SAVE TRAN prestamos
  -- Mark a save point

  SELECT 'After SAVE TRAN prestamos', @@TRANCOUNT
  -- The value of @@TRANCOUNT is still 2

  ROLLBACK TRAN depositos

  SELECT 'After ROLLBACK TRAN depositos', @@TRANCOUNT
```

Puntos de verificación

… ejemplo

```
    -- The value of @@TRANCOUNT is still 2

    SELECT TOP 5 customer_name FROM deposit

IF (@@TRANCOUNT > 0)
BEGIN
  ROLLBACK TRAN

  SELECT 'AFTER ROLLBACK TRAN', @@TRANCOUNT
  /* The value of @@TRANCOUNT is 0 because
     ROLLBACK TRAN always rolls back all transactions and
     sets @@TRANCOUNT to 0 */
END

SELECT TOP 5 customer_name FROM deposit
```

Manejo de errores

`@@ERROR`

- `id` del último error
- Éxito → `@@ERROR = 0`
- Fracaso → `@@ERROR > 0`
- Para determinar si una sentencia se ejecuta exitosamente, entonces es necesario verificar el valor de `@@ERROR` inmediatamente después que dicha sentencia se ejecutó

Manejo de errores

ejemplo 1 …

```
CREATE PROCEDURE addTitle(@title_id VARCHAR(6),
  @au_id VARCHAR(11), @title VARCHAR(20),
  @title_type CHAR(12))
AS
BEGIN TRAN
  INSERT titles(title_id, title, type)
  VALUES (@title_id, @title, @title_type)

  IF (@@ERROR <> 0)
  BEGIN
    PRINT '¡Ocurrió un error no previsto!'
    ROLLBACK TRAN
    RETURN 1
  END
```

Manejo de errores

… ejemplo 1

```
  INSERT titleauthor(au_id, title_id)
  VALUES (@au_id, @title_id)

  IF (@@ERROR <> 0)
  BEGIN
    PRINT '¡Ocurrió un error no previsto!'
    ROLLBACK TRAN
    RETURN 1
  END

COMMIT TRAN

RETURN 0
```

¿Problema?

Manejo de errores

ejemplo 2 …

```
CREATE PROCEDURE addTitle(@title_id VARCHAR(6),
  @au_id VARCHAR(11), @title VARCHAR(20),
  @title_type CHAR(12))
AS
BEGIN TRAN
  INSERT titles(title_id, title, type)
  VALUES (@title_id, @title, @title_type)

  IF (@@ERROR <> 0) GOTO ERR_HANDLER

  INSERT titleauthor(au_id, title_id)
  VALUES (@au_id, @title_id)

  IF (@@ERROR <> 0) GOTO ERR_HANDLER
```

Manejo de errores

… ejemplo 2 …

```
COMMIT TRAN

RETURN 0

ERR_HANDLER:
   PRINT '¡Ocurrió un error no previsto!'
   ROLLBACK TRAN
   RETURN 1
```

Concurrencia, conflictos y schedules

Control de concurrencia

- The problem of synchronizing concurrent transactions such that the consistency of the database is maintained while, at the same time, maximum degree of concurrency is achieved.
- Anomalies:
  - Lost updates
    - The effects of some transactions are not reflected on the database
  - Inconsistent retrievals
    - A transaction, if it reads the same data item more than once, should always read the same value

Concurrencia, conflictos y schedules

Schedule (or execution history)

- An order in which the operations of a set of transactions are executed
- A history (schedule) can be defined as a partial order over the operations of a set of transactions

T_1:	T_2:	T_3:
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

$$H_1=\{W_2(x),R_1(x),\ R_3(x),W_1(x),C_1,W_2(y),R_3(y),R_2(z),C_2,R_3(z),C_3\}$$

Concurrencia, conflictos y schedules

Formalization of History

A complete history over a set of transactions $T=\{T_1, \ldots, T_n\}$ is a partial order $H_t^c = H_c(T) = \{\Sigma_T, \prec_H\}$ where

➊ $\Sigma_T = \bigcup_i \Sigma_i$, for $i = 1, 2, \ldots, n$

➋ $\prec_H \supseteq \bigcup_i \prec_{T_i}$, for $i = 1, 2, \ldots, n$

➌ For any two conflicting operations $O_{ij}, O_{kl} \in \Sigma_T$, either $O_{ij} \prec_H O_{kl}$ or $O_{kl} \prec_H O_{ij}$

Concurrencia, conflictos y schedules

Formalization of History

Given three transactions

T_1:	T_2:	T_3:
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

$H_1=\{W_2(x),R_1(x), R_3(x),W_1(x),C_1,W_2(y),R_3(y),R_2(z),C_2,R_3(z),C_3\}$

Concurrencia, conflictos y schedules

Formalization of History

A possible schedule is given as the DAG

$$\begin{tikzcd}
R_1(x) \arrow[d] & W_2(x) \arrow[l] \arrow[r] \arrow[d] & R_3(x) \arrow[d] \\
W_1(x) \arrow[d] & W_2(y) \arrow[r] \arrow[d] & R_3(y) \arrow[d] \\
C_1 & R_2(z) \arrow[r] \arrow[d] & R_3(z) \arrow[d] \\
 & C_2 & C_3
\end{tikzcd}$$

Concurrencia, conflictos y schedules

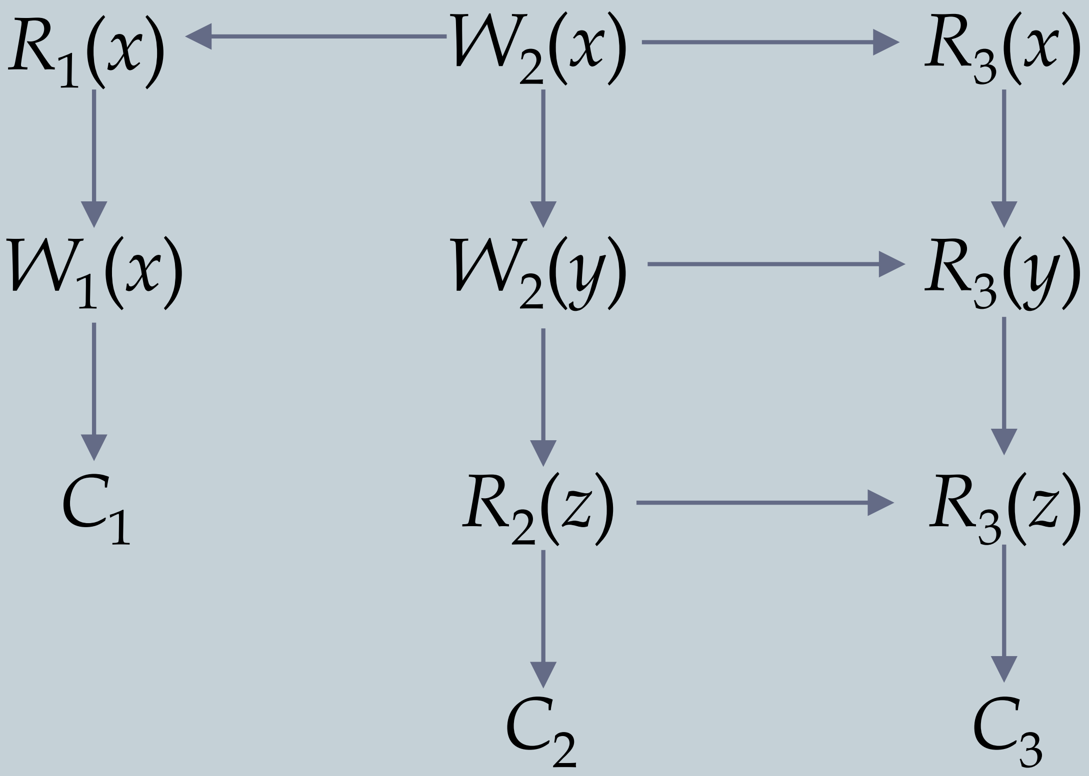
Schedule: *definition*

A schedule is a prefix of a complete schedule such that only some of the operations and only some of the ordering relationships are included

T1:	T2:	T3:
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

Concurrencia, conflictos y schedules

Schedule: *definition*



$R_1(x)$ $W_2(x)$ $R_3(x)$

$W_1(x)$ $W_2(y)$ $R_3(y)$

C_1 $R_2(z)$ $R_3(z)$

C_2 C_3

A complete history H^c for the transactions T_1, T_2, T_3

Concurrencia, conflictos y schedules

Schedule: *definition*

$$\begin{tikzcd}
R_1(x) & W_2(x) \arrow[l] \arrow[r] \arrow[d] & R_3(x) \arrow[d] \\
 & W_2(y) \arrow[r] \arrow[d] & R_3(y) \arrow[d] \\
 & R_2(z) \arrow[r] & R_3(z)
\end{tikzcd}$$

A history H for the transactions T_1, T_2, T_3

Concurrencia, conflictos y schedules

Serial history

- All the actions of a transaction occur consecutively
- No interleaving of transaction operations
- The serial execution of a set of transactions maintains the consistency of the database
- If each transaction is consistent (obeys integrity rules), then the database is guaranteed to be consistent at the end of executing a serial history

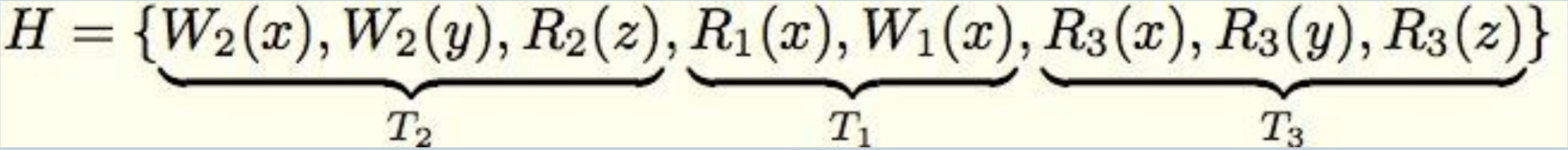
Concurrencia, conflictos y schedules

Serial history

T_1: Read(x)
Write(x)
Commit

T_2: Write(x)
Write(y)
Read(z)
Commit

T_3: Read(x)
Read(y)
Read(z)
Commit



$$H = \{\underbrace{W_2(x), W_2(y), R_2(z)}_{T_2}, \underbrace{R_1(x), W_1(x)}_{T_1}, \underbrace{R_3(x), R_3(y), R_3(z)}_{T_3}\}$$

$$T_2 \rightarrow T_1 \rightarrow T_3$$

$$T_2 \prec_H T_1 \prec_H T_3$$

Concurrencia, conflictos y schedules

equivalent histories

- Equivalent histories

  Histories: H_1 and H_2

  Set of transactions: T

  H_1 and H_2 are *equivalent* if they have the same effect on the database

- Formal definition

  Two histories, H_1 and H_2, defined over the same set of transactions T, are said to be equivalent if for each pair of conflicting operations O_{ij} and O_{kl} $(i \neq k)$, whenever $O_{ij} \prec_{H_1} O_{kl}$, then $O_{ij} \prec_{H_2} O_{kl}$

Concurrencia, conflictos y schedules

Serializable history

- Transactions execute concurrently, but the net effect of the resulting history upon the database is equivalent to some serial history
- Equivalent with respect to what?
  - **Conflict equivalence**: the relative order of execution of the conflicting operations belonging to unaborted transactions in two histories are the same
  - **Conflicting operations**: two incompatible operations (e.g., Read and Write) conflict if they both access the same data item
    - Incompatible operations of each transaction is assumed to conflict; do not change their execution orders
    - If two operations from two different transactions conflict, the corresponding transactions are also said to conflict

Concurrencia, conflictos y schedules

H' is conflict equivalent to H

$$H' = \{W_2(x), R_1(x), W_1(x), R_3(x), W_2(y), R_3(y), R_2(z), R_3(z)\}$$

$$H = \{\underbrace{W_2(x), W_2(y), R_2(z)}_{T_2}, \underbrace{R_1(x), W_1(x)}_{T_1}, \underbrace{R_3(x), R_3(y), R_3(z)}_{T_3}\}$$

Concurrencia, conflictos y schedules

Serializable history

- A history H is said to be *serializable* if and only if it is conflict equivalent to a serial history
- Serializability roughly corresponds to degree 3 consistency
- Serializability is also known as *conflict-based serializability*
- *H'* is serializable since it is equivalent to the serial history *H*

$$H' = \{W_2(x), R_1(x), W_1(x), R_3(x), W_2(y), R_3(y), R_2(z), R_3(z)\}$$

$$H = \{\underbrace{W_2(x), W_2(y), R_2(z)}_{T_2}, \underbrace{R_1(x), W_1(x)}_{T_1}, \underbrace{R_3(x), R_3(y), R_3(z)}_{T_3}\}$$

Concurrencia, conflictos y schedules

Serializable history

T_1:	T_2:	T_3:
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

The following are not conflict equivalent

$$H_s=\{W_2(x),W_2(y),R_2(z),R_1(x),W_1(x),R_3(x),R_3(y),R_3(z)\}$$

$$H_1=\{W_2(x),R_1(x), R_3(x),W_1(x),W_2(y),R_3(y),R_2(z),R_3(z)\}$$

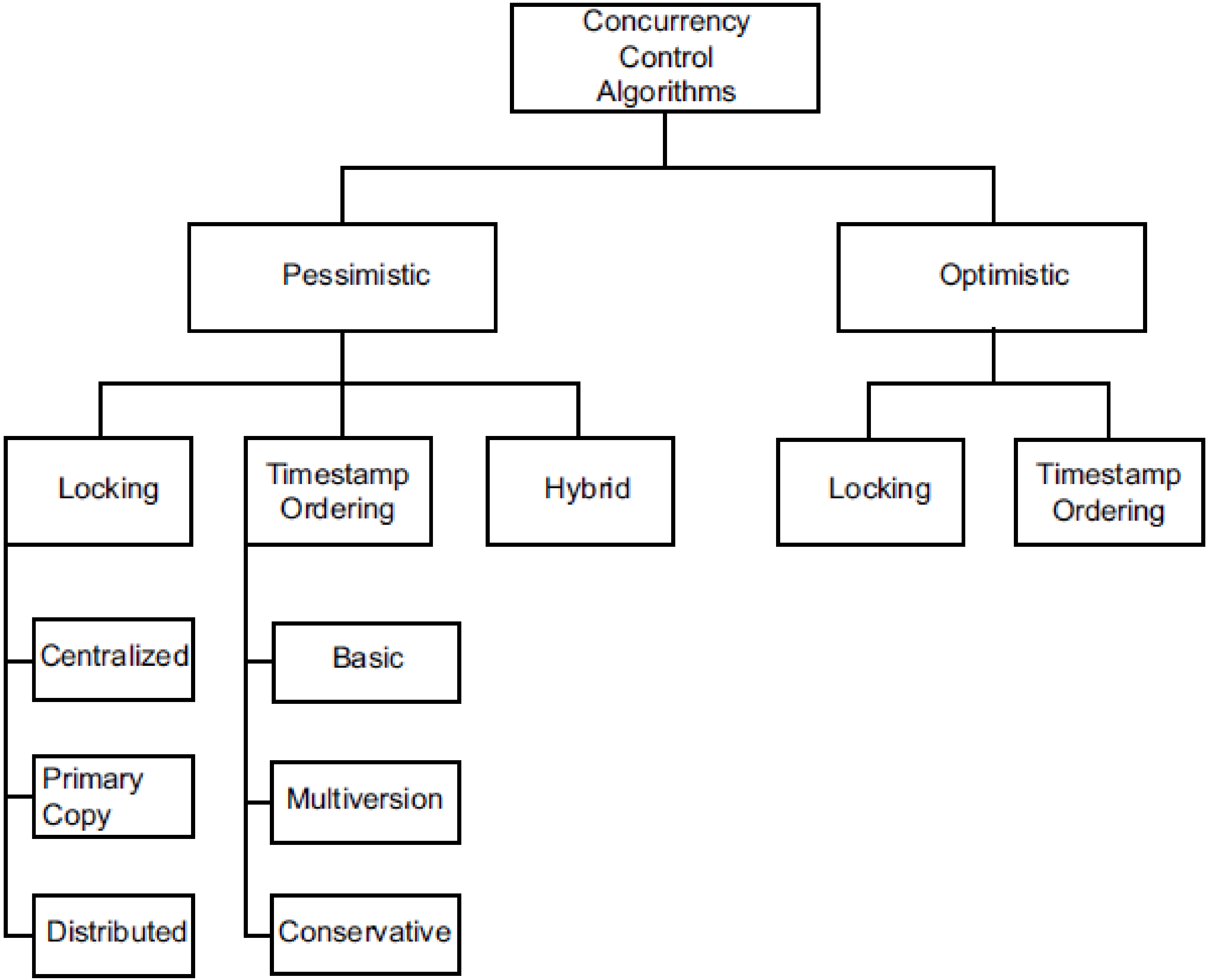
The following are conflict equivalent; therefore H_2 is *serializable*

$$H_s=\{W_2(x),W_2(y),R_2(z),R_1(x),W_1(x),R_3(x),R_3(y),R_3(z)\}$$

$$H_2=\{W_2(x),R_1(x),W_1(x),R_3(x),W_2(y),R_3(y),R_2(z),R_3(z)\}$$

Algoritmos para el control de la concurrencia

classification of concurrency control algorithms



Algoritmos para el control de la concurrencia

Locking-Based Algorithms

- Transactions indicate their intentions by requesting locks from the scheduler (called lock manager)
- Types of locks
  - read lock (*rl*) [also called shared lock]
  - write lock (*wl*) [also called exclusive lock]
- Transaction T_i
  - $rl_i(X)$
  - $wl_i(X)$
- *Compatible modes*

  Two lock modes are compatible if two transactions that access the same date item can obtain these locks on that data item at the same time

Algoritmos para el control de la concurrencia

Locking-Based Algorithms

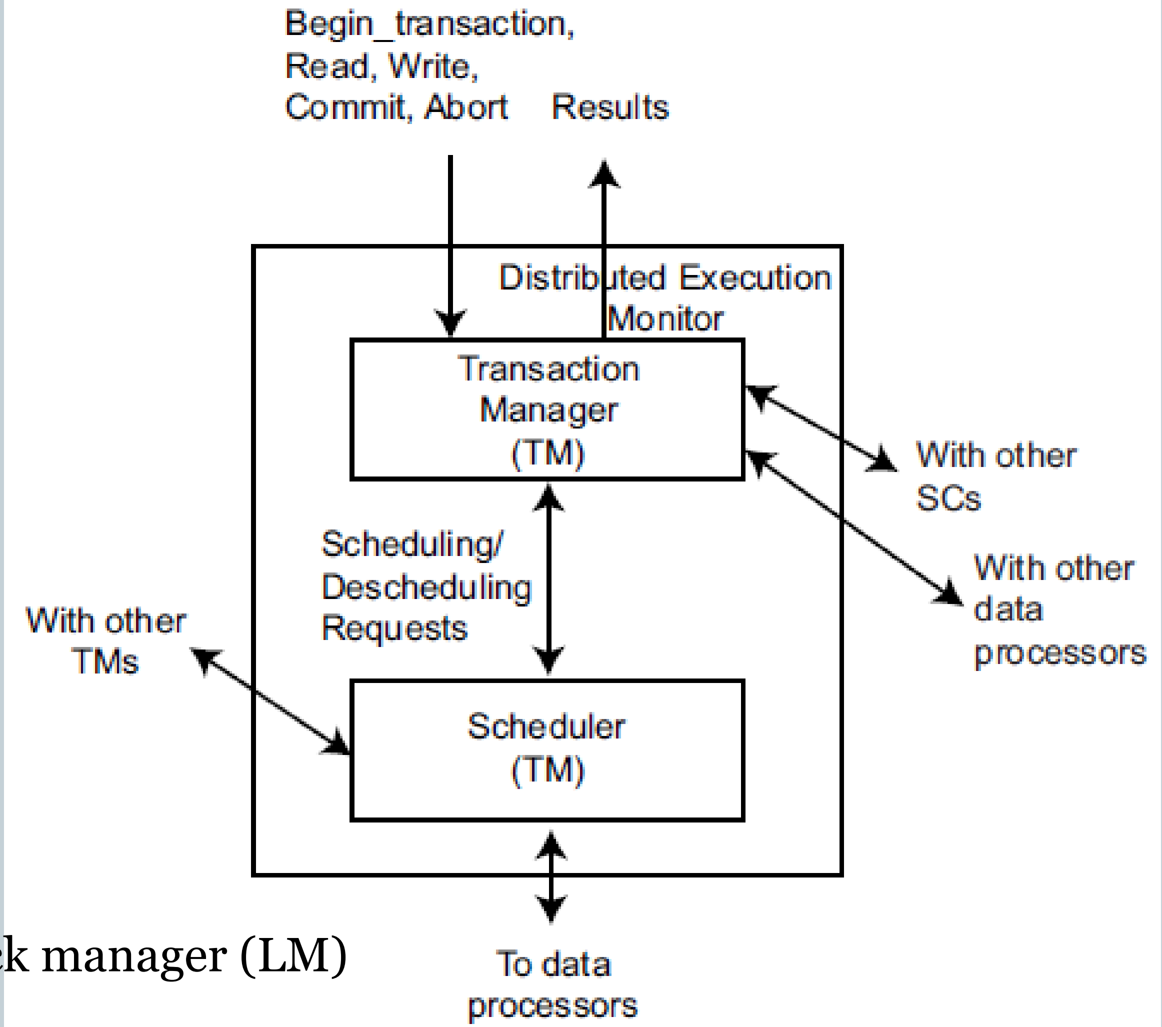
- Read locks and write locks conflict (because Read and Write operations are incompatible

	rl	*wl*
rl	yes	no
wl	no	no

- Locking works nicely to allow concurrent processing of transactions

Algoritmos para el control de la concurrencia

Locking-Based Algorithms



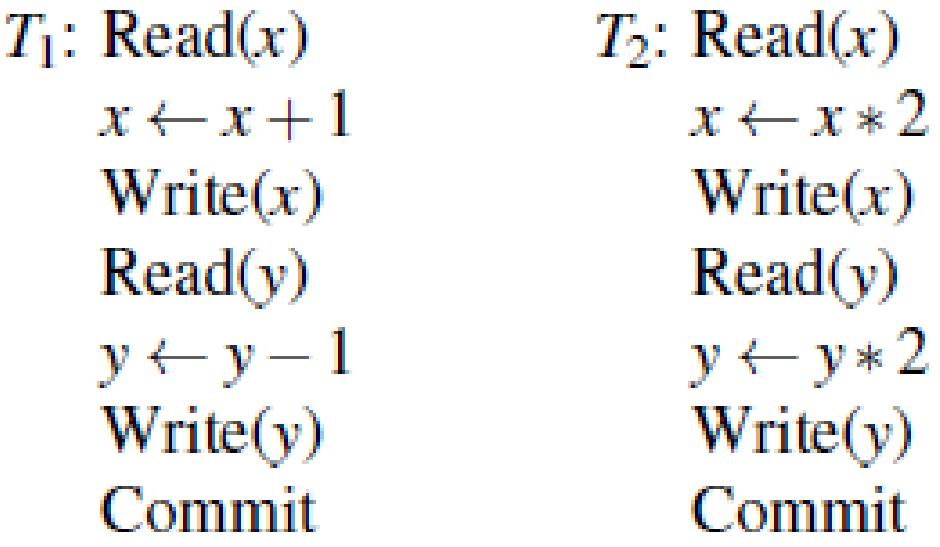
Scheduler = lock manager (LM)

Algoritmos para el control de la concurrencia

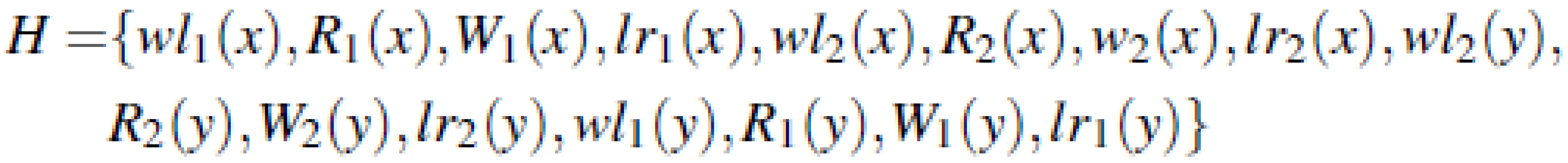
Locking-Based Algorithms

To generate serializable histories, the locking and releasing operations of transactions also need to be coordinated

$lr_i(z)$: indicates the release of the Lock on z that Transaction T_i holds



T_1: Read(x)	T_2: Read(x)
$x \leftarrow x + 1$	$x \leftarrow x * 2$
Write(x)	Write(x)
Read(y)	Read(y)
$y \leftarrow y - 1$	$y \leftarrow y * 2$
Write(y)	Write(y)
Commit	Commit



$$H = \{wl_1(x), R_1(x), W_1(x), lr_1(x), wl_2(x), R_2(x), w_2(x), lr_2(x), wl_2(y), R_2(y), W_2(y), lr_2(y), wl_1(y), R_1(y), W_1(y), lr_1(y)\}$$

Algoritmos para el control de la concurrencia

Locking-Based Algorithms

- Is *H* a serializable history?
- No
- Problem with *H*

  It permits transactions to interfere with one another, resulting in the loss of isolation and atomicity

Locking-Based Algorithms

Two-Phase Locking (2PL)

- The two-phase locking rule states that no transaction should request a lock after it releases one of its locks
- A transaction should not release a lock until it is certain that it will not request another lock
- 2PL algorithm execute transactions in two phases
  - Each transaction has a **growing phase**, where is obtains locks and access data items, and
  - A **shrinking phase**, during which it releases locks

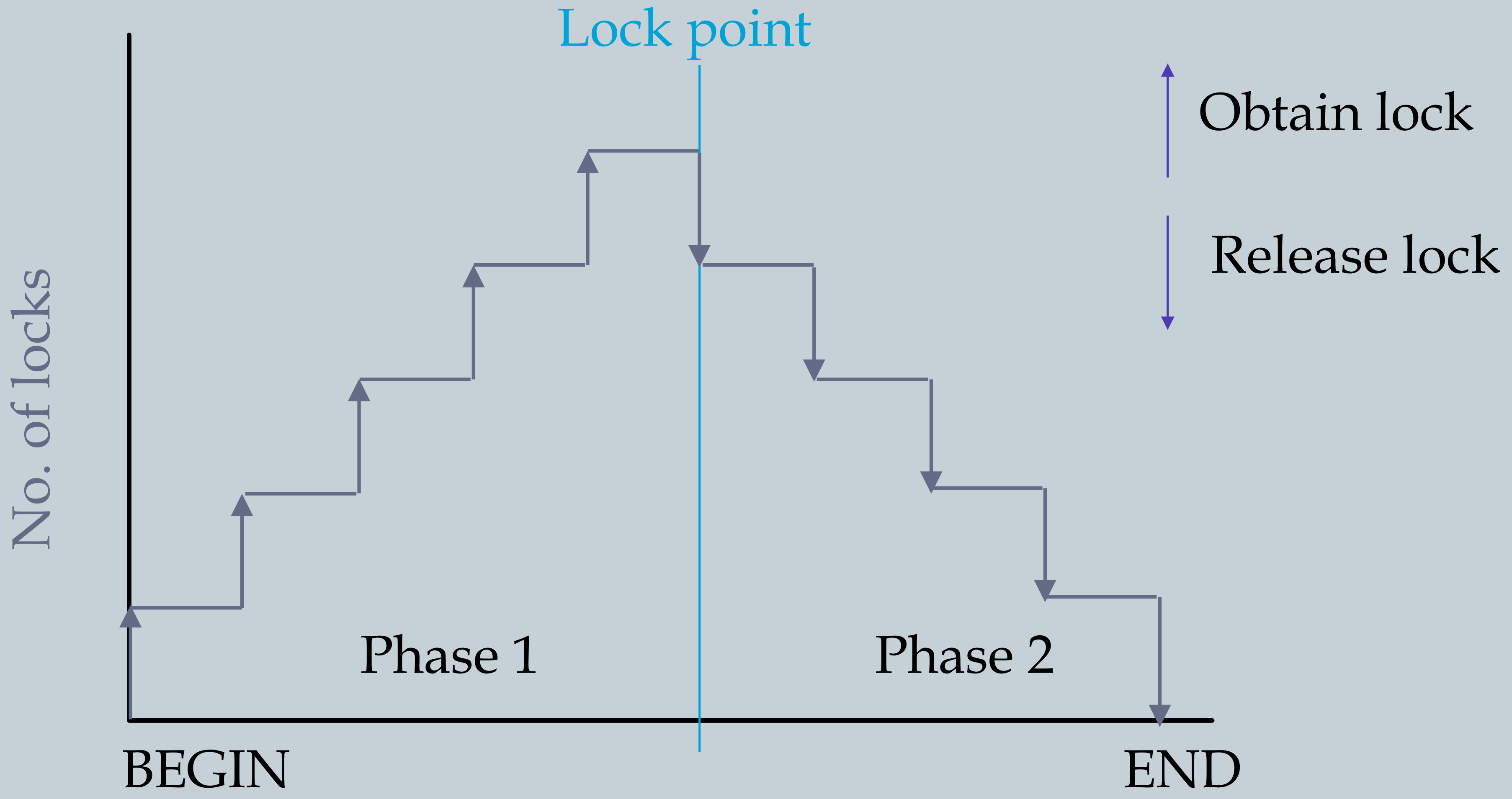
Locking-Based Algorithms

Two-Phase Locking (2PL)

1. A transaction locks an object before using it
2. When an object is locked by another transaction, the requesting transaction must wait
3. When a transaction releases a lock, it may not request another lock

Locking-Based Algorithms

Two-Phase Locking (2PL)



Locking-Based Algorithms

Two-Phase Locking (2PL): implementation problems

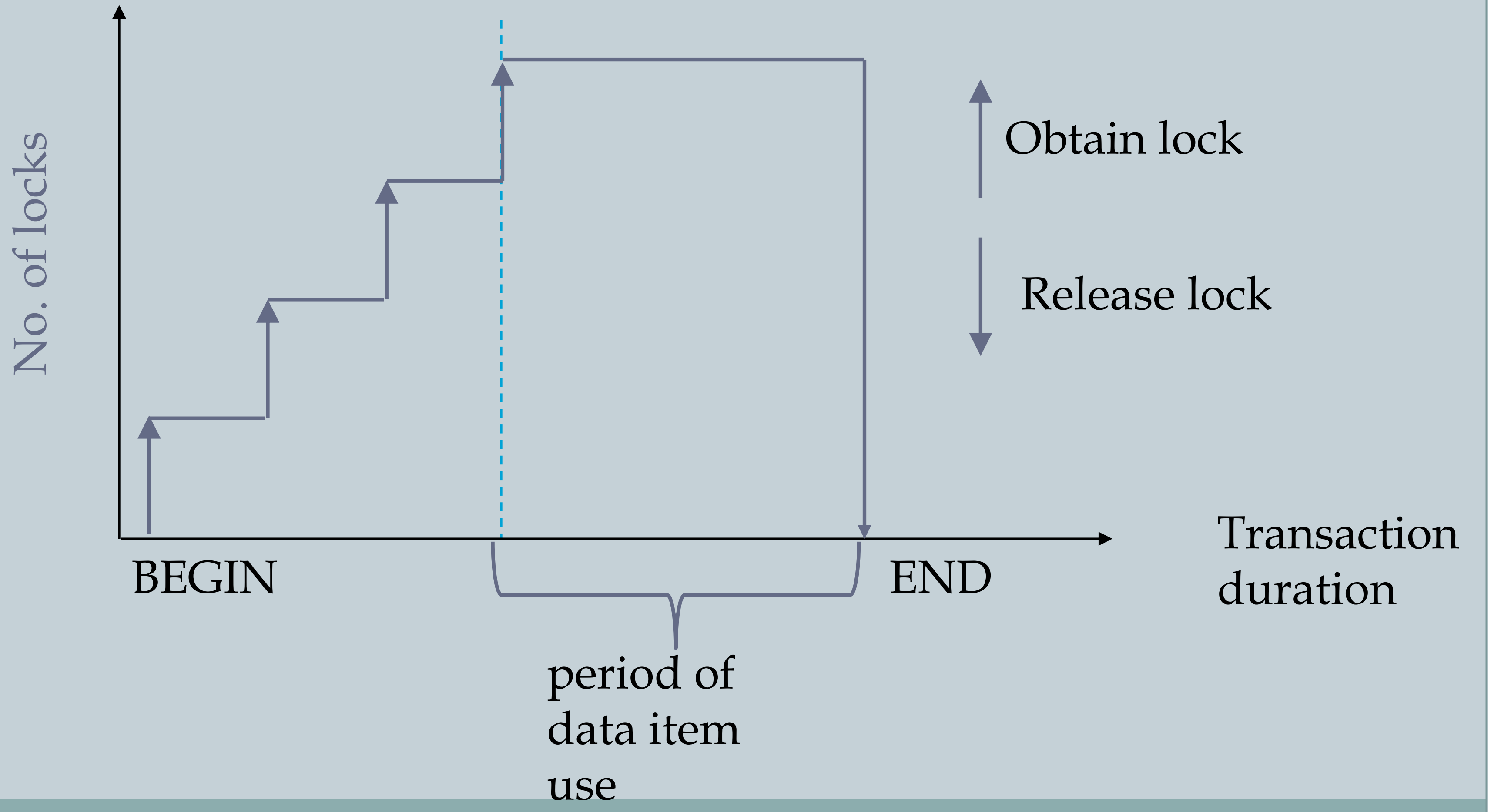
1. The lock manager has to know that the transaction has obtained all its locks and will not need to lock another data item
2. The lock manager needs to know that the transaction no longer needs to access the data item in question, so that the lock can be released
3. If the transaction aborts after it releases a lock, it may cause other transactions that may have accessed the unlocked data item to abort as well – *cascading aborts*

⇒ strict two-phase locking: releases all the locks together when the transaction terminates (commits or aborts)

Locking-Based Algorithms

Strict 2PL

Hold locks until the end.



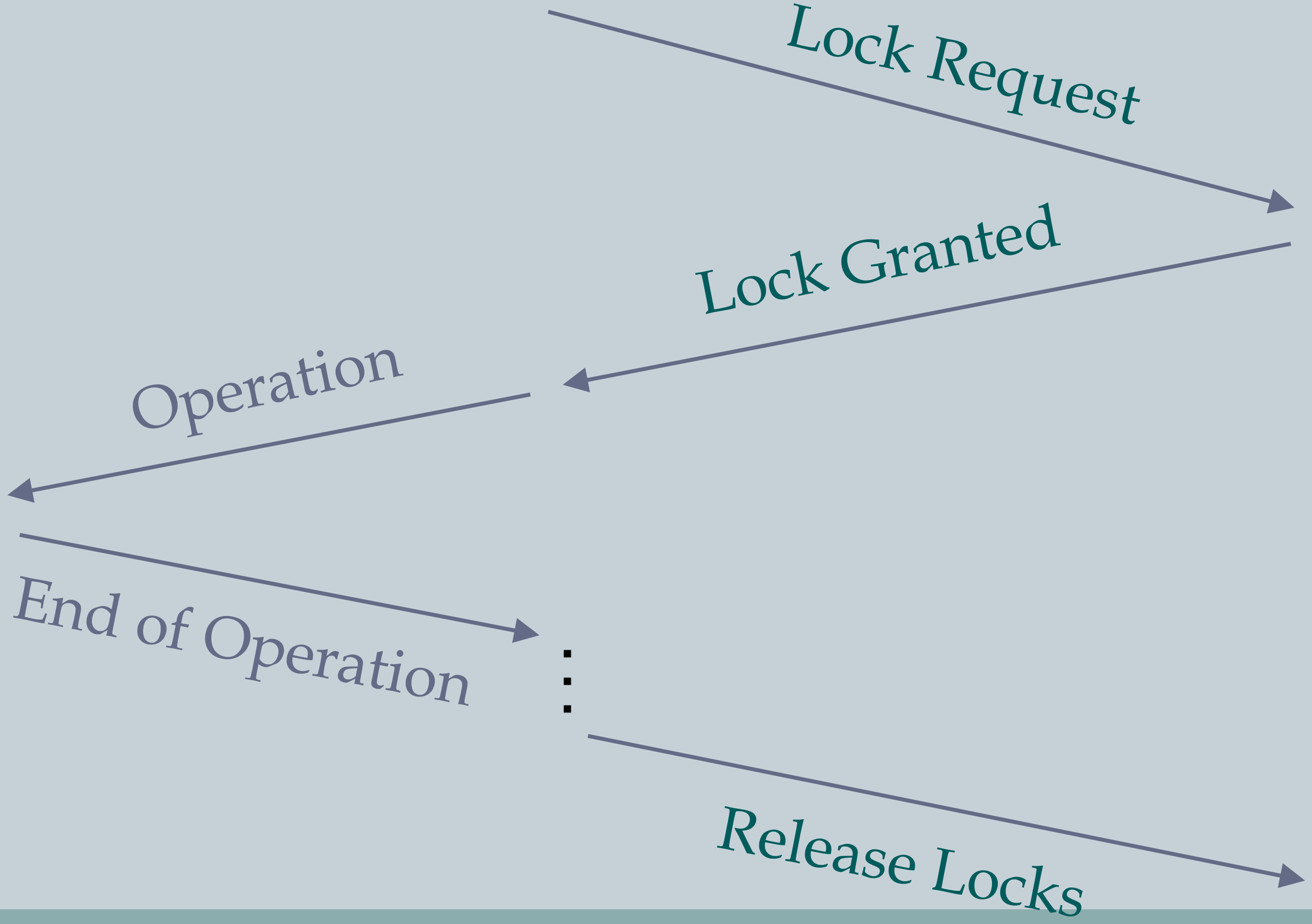
Locking-Based Algorithms

Centralized 2PL (C2PL)

- The 2PL algorithm can be extended to the distributed DBMS environment
- There is only one 2PL scheduler in the distributed system
- Lock requests are issued to the central scheduler
- This means that only one of the sites has a lock manager; the transaction managers at the other sites communicate with it rather than with their own lock managers

Locking-Based Algorithms

C2PL



Data Processors at participating sites

Coordinating TM

Central Site TM

Lock Request

Lock Granted

Operation

End of Operation

⋮

Release Locks

Algoritmos para el control de la concurrencia

Timestamp-based concurrency control algorithms

- They do not attempt to maintain serializability by mutual exclusion
- They select a serialization order and execute transactions accordingly
- To establish this ordering, the transaction manager assigns each transation T_i a unique timestamps, $ts(T_i)$, at its initiation
- Timestamp?
  - A simple identifier that serves to identify each transaction uniquely and is used for ordering
  - Properties: uniqueness, monotonicity

Algoritmos para el control de la concurrencia

Timestamp-based concurrency control algorithms

Methods to assign timestamps

- Use a global (system-wide) monotonically increasing counter

  Problem: the maintenance of global counters

- Each site autonomously assigns timestamps based on its local counter
  - (local counter value, site identifier)
  - If each system can access its own system clock, it is possible to use system clock values instead of counter values

Algoritmos para el control de la concurrencia

TO Rule

Given two conflicting operations O_{ij} and O_{kl} belonging, respectively, to transactions T_i and T_k, O_{ij} is executed before O_{kl} if and only if $ts(T_i) < ts(T_k)$

T_i: older transaction

T_k: younger transaction

A timestamps ordering scheduler is guaranteed to generate serializable histories

Algoritmos para el control de la concurrencia

Timestamp Ordering

1. Transaction T_i is assigned a globally unique timestamp $ts(T_i)$
2. Transaction manager attaches the timestamp to all operations issued by the transaction
3. Each data item is assigned a write timestamp (*wts*) and a read timestamp (*rts*):
   - rts(x) = largest timestamp of any read on x
   - wts(x) = largest timestamp of any write on x
4. Conflicting operations are resolved by timestamp order

Algoritmos para el control de la concurrencia

Basic Timestamp Ordering Algorithm

- A transaction one of whose operations is rejected by a scheduler is restarted by the transaction manager with a new timestamp
- This ensures that the transaction has a chance to execute in its next try
- Since the transactions never wait while they hold access rights to data items, the basic TO algorithm never causes deadlocks
- The penalty of deadlock freedom is potential restart of a transaction numerous times
- There is a alternative to the basic TO algorithm that reduces the number of restarts

Algoritmos para el control de la concurrencia

Basic Timestamp Ordering Algorithm

- The Timestamps Ordering (TO) scheduler first receives $W_i(x)$ and then receives $W_j(x)$, where $ts(T_i) < ts(T_j)$
- The scheduler would accept both operations and pass them on to the data processor
- The result of these two operations is that $wts(x) = ts(T_j)$ and we expect the effect of $W_j(x)$ to be represented in the database
- If the data processor does not execute them in that order, the effects on the database will be wrong

Algoritmos para el control de la concurrencia

Strict Timestamp Ordering Algorithm

If $W_i(x)$ is accepted and released to the data processor, the scheduler delays all $R_j(x)$ and $W_j(x)$ operations (for all T_j) until T_i terminates (commit or aborts)

Algoritmos para el control de la concurrencia

Timestamp Ordering: Basic TO Scheduler Algorithm

```
for $R_i(x)$

if $ts(T_i) < wts(x)$
then reject $R_i(x)$
else accept $R_i(x)$
    $rts(x) \leftarrow ts(T_i)$
```

```
for $W_i(x)$

if $ts(T_i) < rts(x)$ and $ts(T_i) < wts(x)$
then reject $W_i(x)$
else accept $W_i(x)$
    $wts(x) \leftarrow ts(T_i)$
```

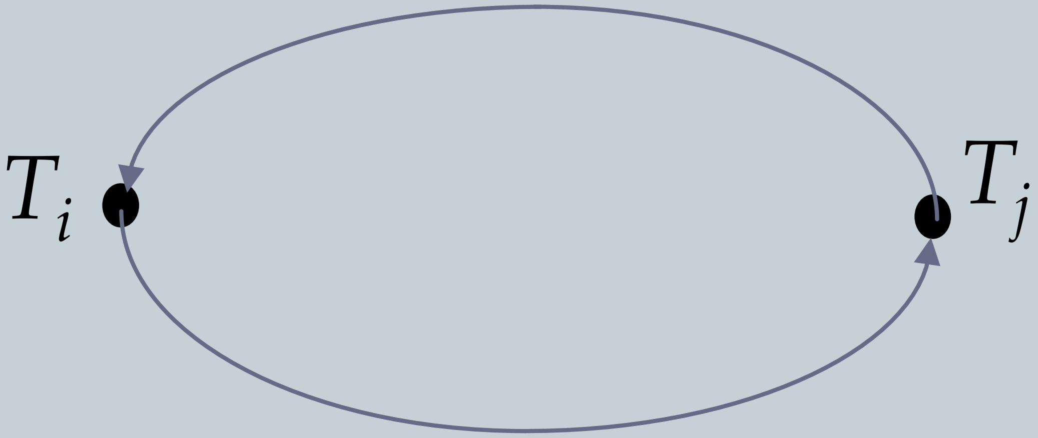
Algoritmos para el control de la concurrencia

Deadlock

- A *deadlock* can occur because transactions wait for one another
- A deadlock situation is a set of requests that can never be granted by the consistency control mechanism
- A transaction is deadlocked if it is blocked and will remain blocked until there is intervention
- Locking-based CC algorithms may cause deadlocks
- TO-based algorithms that involve waiting may cause deadlocks
- Wait-For Graph (WFG)
  - If transaction T_i waits for another transaction T_j to release a lock on an entity, then $T_i \rightarrow T_j$ in WFG

Algoritmos para el control de la concurrencia

Deadlock



Deadlock

Deadlock Management: methods

- Ignore

  Let the application programmer deal with it, or restart the system

- Prevention
  - Guaranteeing that deadlocks can never occur in the first place
  - Check transaction when it is initiated
  - Data items accessed: predeclared, pb: difficult task
  - Requires no run time support
- Avoidance
  - Detecting potential deadlocks in advance and taking action to insure that deadlock will not occur
  - Simple approach: order the resources and access them in that order
  - Requires no run time support
- Detection and Recovery
  - Allowing deadlocks to form and then finding and breaking them. As in the avoidance scheme
  - requires run time support

Deadlock

Deadlock Management: Deadlock Prevention

- All resources which may be needed by a transaction must be predeclared
  - The system must guarantee that none of the resources will be needed by an ongoing transaction
  - Resources must only be reserved, but not necessarily allocated a priori
  - Unsuitability of the scheme in database environment
  - Suitable for systems that have no provisions for undoing processes
- Evaluation
  - Reduced concurrency due to preallocation
  - Evaluating whether an allocation is safe leads to added overhead
  - – Difficult to determine (partial order)
  - \+ No transaction rollback or restart is involved

Deadlock

Deadlock Avoidance

- Transactions are not required to request resources a priori
- Transactions are allowed to proceed unless a requested resource is unavailable
- In case of conflict, transactions may be allowed to wait for a fixed time interval
- Order either the data items or the sites and always request locks in that order
- More attractive than prevention in a database environment

Deadlock

Deadlock Avoidance – Wait-Die Algorithm

- If T_i requests a lock on a data item which is already locked by T_j, then T_i is permitted to wait iff $ts(T_i)<ts(T_j)$
- If $ts(T_i)>ts(T_j)$, then T_i is aborted and restarted with the same timestamp
  - **if** $ts(T_i)<ts(T_j)$ **then** T_i waits **else** T_i dies
  - non-preemptive: T_i never preempts T_j
  - prefers younger transactions

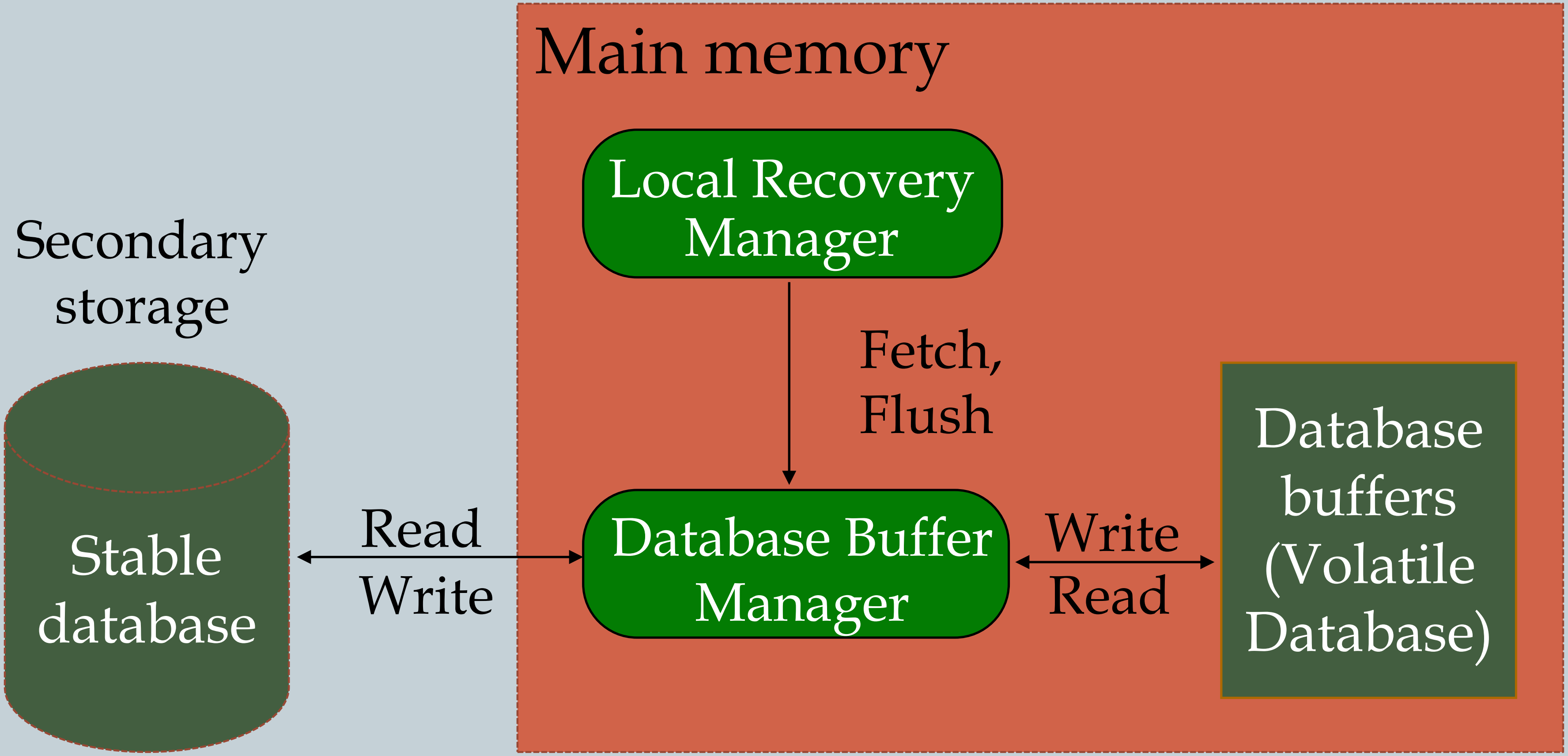
Deadlock

Deadlock Avoidance – Wound-Wait Algorithm

- If T_i requests a lock on a data item which is already locked by T_j, then T_i is permitted to wait iff $ts(T_i) > ts(T_j)$
- If $ts(T_i) < ts(T_j)$, then T_j is aborted and the lock is granted to T_i
    - **if** $ts(T_i)<ts(T_j)$ **then** T_j is wounded **else** T_i waits
    - preemptive: T_i preempts T_j if it is younger
    - prefers older transactions

Administración de la recuperación local

arquitectura



Main memory

Local Recovery Manager

Fetch, Flush

Secondary storage

Stable database

Read Write

Database Buffer Manager

Write Read

Database buffers (Volatile Database)

Page: unit of storage and acces of the stable database

Administración de la recuperación local

recovery information

- System failures ⇒ the volatile database is lost
- *Recovery information*: the information that the DBMS maintains about its state at the time of the failure in order to able to bring the database to the state that it was when the failure ocurred
- The recovery information that the system maintains is dependent on the method of executing updates
  - In-place updating
  - Out-of-place updating

Administración de la recuperación local

arquitectura

- In-place update
  - Each update causes a change in one or more data values on pages in the database buffers
  - Database log
  - The most common update technique

- Out-of-place update

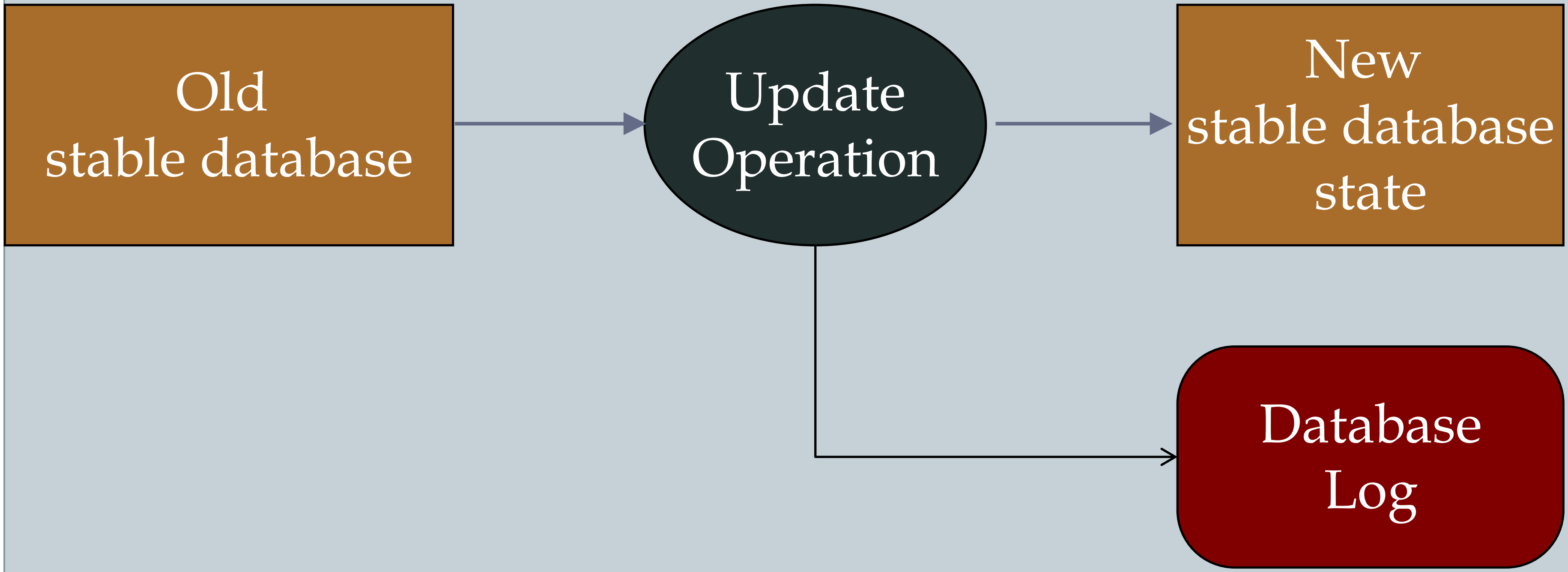
  Each update causes the new value(s) of data item(s) to be stored separate from the old value(s)

Administración de la recuperación local

in-place update recovery information

Database log

Every action of a transaction must not only perform the action, but must also write a log record to an append-only file



Old stable database

Update Operation

New stable database state

Database Log

Administración de la recuperación local

logging

The log contains information used by the recovery process to restore the consistency of a system. This information may include

- transaction identifier
- type of operation (action)
- items accessed by the transaction to perform the action
- old value (state) of item (before image)
- new value (state) of item (after image)

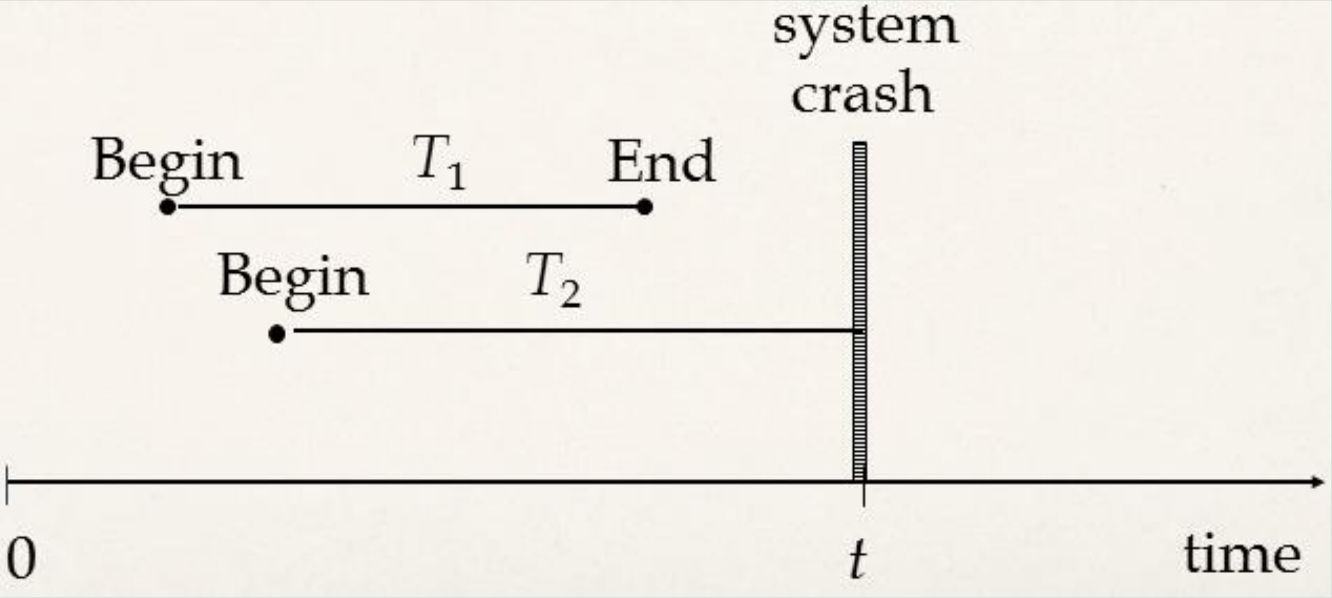
…

Administración de la recuperación local

why logging?

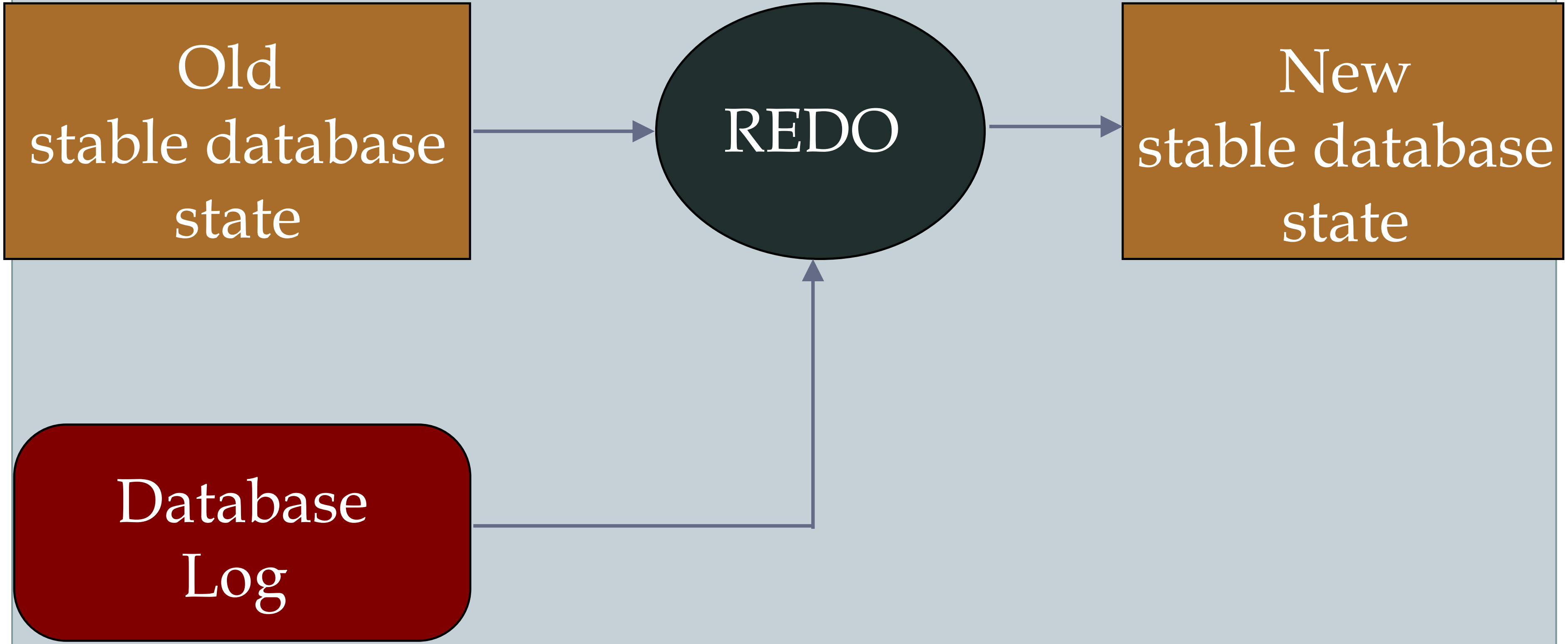
Upon recovery:

- all of T_1's effects should be reflected in the database (REDO if necessary due to a failure)
- none of T_2's effects should be reflected in the database (UNDO if necessary)



Administración de la recuperación local

REDO Protocol



Old
stable database
state

REDO

New
stable database
state

Database
Log

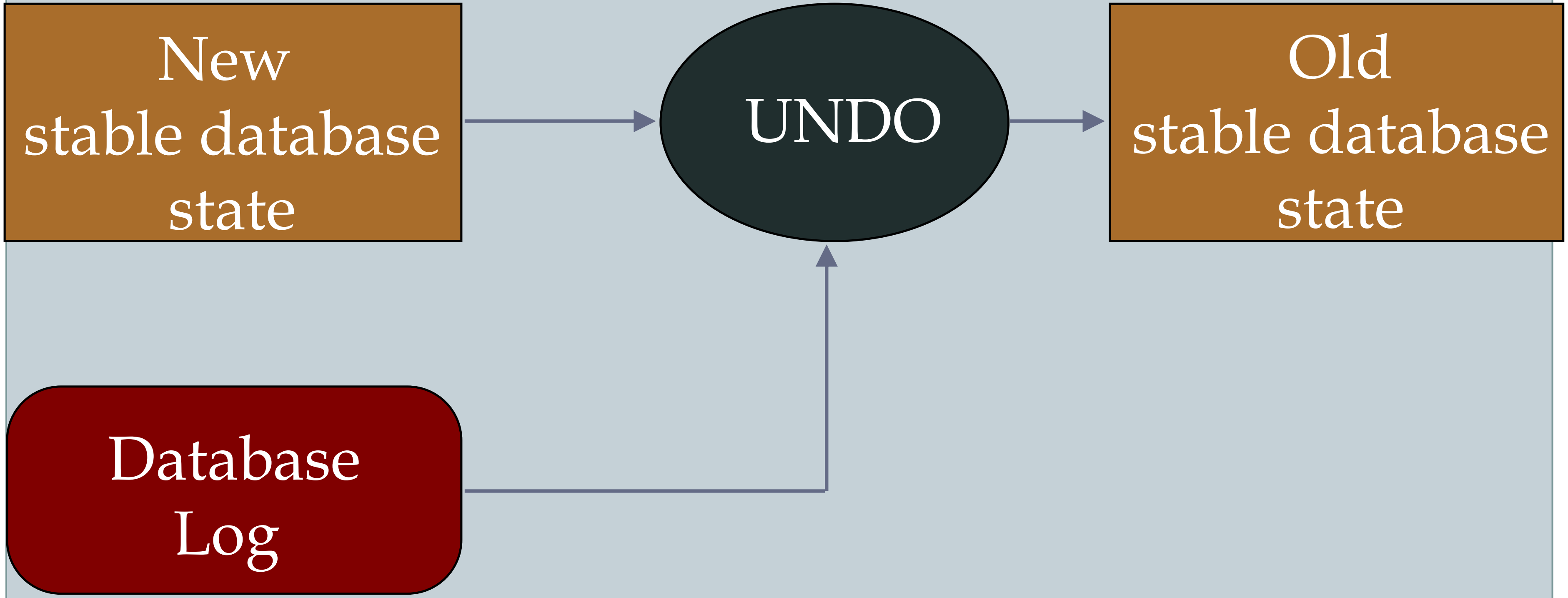
Administración de la recuperación local

REDO Protocol

- REDO'ing an action means performing it again
- The REDO operation uses the log information and performs the action that might have been done before, or not done due to failures
- The REDO operation generates the new image

Administración de la recuperación local

UNDO Protocol



New stable database state

UNDO

Old stable database state

Database Log

Administración de la recuperación local

UNDO Protocol

- UNDO'ing an action means to restore the object to its before image
- The UNDO operation uses the log information and restores the old value of the object

Administración de la recuperación local

When to write log records into stable store

Assume a transaction *T* updates a page *P*

- Fortunate case
  - System writes P in stable database
  - System updates stable log for this update
  - SYSTEM FAILURE OCCURS!... (before T commits)

  We can recover (undo) by restoring *P* to its old state by using the log
- Unfortunate case
  - System writes P in stable database
  - SYSTEM FAILURE OCCURS!... (before stable log is updated)

  We cannot recover from this failure because there is no log record to restore the old value
- Solution: Write-Ahead Log (WAL) protocol

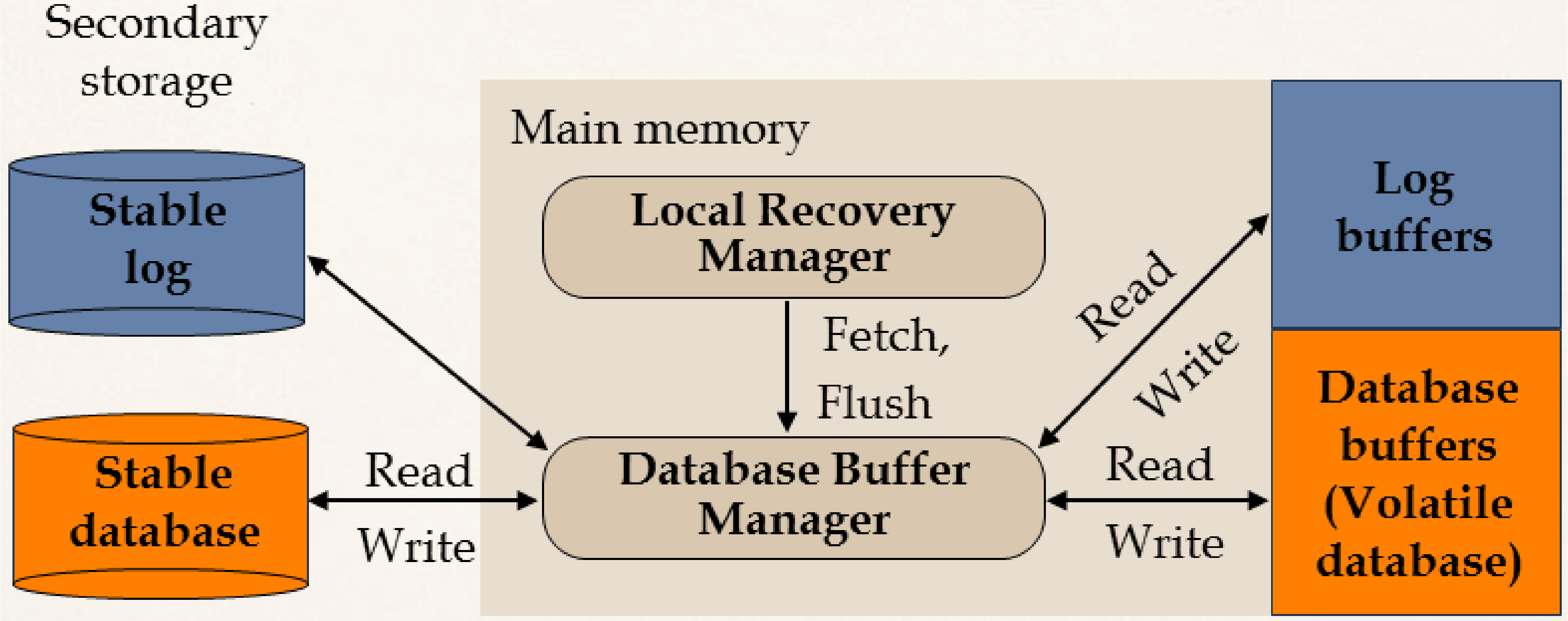
Administración de la recuperación local

Write-Ahead Log protocol

- Notice:
  - If a system crashes before a transaction is committed, then all the operations must be undone. Only need the before images (undo portion of the log)
  - Once a transaction is committed, some of its actions might have to be redone. Need the after images (redo portion of the log)
- WAL protocol:
  1. Before a stable database is updated, the undo portion of the log should be written to the stable log
  2. When a transaction commits, the redo portion of the log must be written to stable log prior to the updating of the stable database

Administración de la recuperación local

logging interface



Administración de la recuperación local

out-of-place update recovery information

- Shadowing
  - When an update occurs, don't change the old page, but create a shadow page with the new values and write it into the stable database
  - Update the access paths so that subsequent accesses are to the new shadow page
  - The old page retained for recovery
- Differential files
  - For each file F maintain
    - a read only part FR
    - a differential file consisting of insertions part DF^+ and deletions part DF^-
    - Thus,$F = (FR \cup DF^+) - DF^-$
  - Updates treated as delete old value, insert new value
  - Periodically, the differential file needs to be merged with the read-only base file

Administración de la recuperación local

execution of commands

Commands to consider:

- begin_transaction
- read
- write
- commit
- abort
- recover

Independent of execution strategy for LRM

Administración de la recuperación local

execution strategies

- Dependent upon
  - Can the buffer manager decide to write some of the buffer pages being accessed by a transaction into stable storage or does it wait for LRM to instruct it?
    - fix/no-fix decision
  - Does the LRM force the buffer manager to write certain buffer pages into stable database at the end of a transaction's execution?
    - flush/no-flush decision
- Possible execution strategies:
  - no-fix/no-flush ⇒ redo/undo algorithm
  - no-fix/flush ⇒ undo/no-redo algorithm
  - fix/no-flush
  - fix/flush

Administración de la recuperación local

execution strategies: No-fix/No-flush

- Abort
  - Buffer manager may have written some of the updated pages into stable database
  - LRM performs transaction undo (or partial undo)
- Commit
  - LRM writes an “end_of_transaction” record into the log.
- Recover
  - For those transactions that have both a “begin_transaction” and an “end_of_transaction” record in the log, a partial redo is initiated by LRM
  - For those transactions that only have a “begin_transaction” in the log, a global undo is executed by LRM

Administración de la recuperación local

execution strategies: No-fix/Flush (undo/no-redo)

- Abort
  - Buffer manager may have written some of the updated pages into stable database
  - LRM performs transaction undo (or partial undo)
- Commit
  - LRM issues a flush command to the buffer manager for all updated pages
  - LRM writes an "end_of_transaction" record into the log
- Recover
  - No need to perform redo since all the updated page are written into the stable database at the commit point
  - Perform global undo: the recovery action initiated by the LRM

Administración de la recuperación local

execution strategies: Fix/No-flush

- The LRM controls the writing of the volatile database pages into a stable storage
- The key is not to permit the buffer manager to write any updated volatile database page into the stable database until at least the transaction commit point: `fix` command (modified version of the `fetch` command)
- Abort
  - None of the updated pages have been written into stable database
  - Release the fixed pages
- Commit
  - LRM writes an "end_of_transaction" record into the log
  - LRM sends an unfix command to the buffer manager for all pages that were previously fixed
- Recover
  - Perform partial redo
  - No need to perform global undo

Administración de la recuperación local

execution strategies: Fix/Flush (no-undo/no-redo)

- The LRM forces the buffer manager to write the updated volatile database pages into the stable database at the commit point – not before and not after
- Abort
  - None of the updated pages have been written into stable database
  - Release the fixed pages
- Commit (the following have to be done atomically)
  - LRM issues a flush command to the buffer manager for all updated pages
  - LRM sends an unfix command to the buffer manager for all pages that were previously fixed
  - LRM writes an “end_of_transaction” record into the log
- Recover
  - No need to do anything

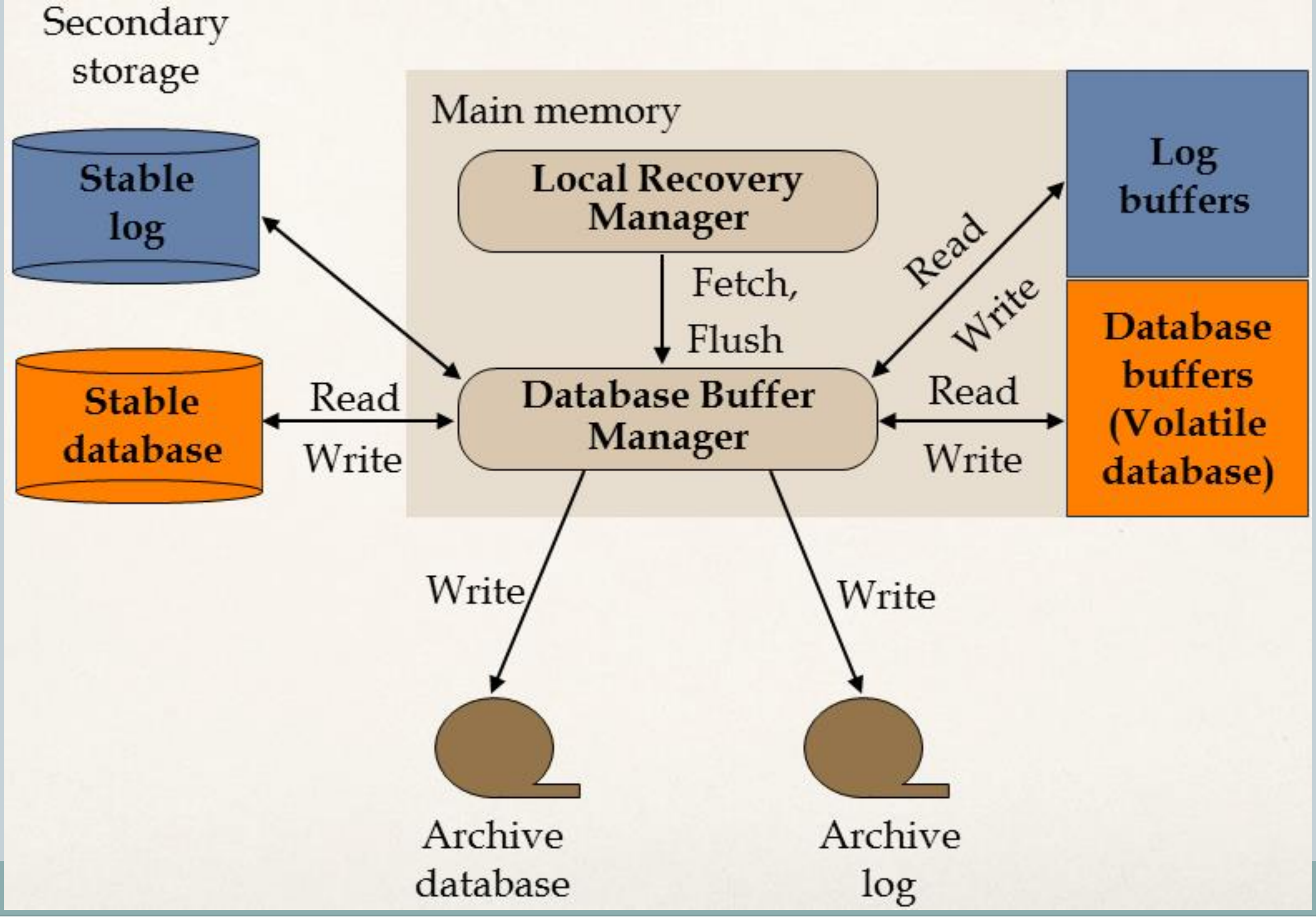
Administración de la recuperación local

checkpoints

- Simplifies the task of determining actions of transactions that need to be undone or redone when a failure occurs
- A *checkpoint* record contains a list of active transactions
- Steps:
  1. Write a begin_checkpoint record into the log
  2. Collect the checkpoint data into the stable storage
  3. Write an end_checkpoint record into the log

Administración de la recuperación local

media failures – full architecture



Deferred update and immediate update

Deferred update	Immediate update
NO-UNDO/REDO	UNDO/REDO

Fin