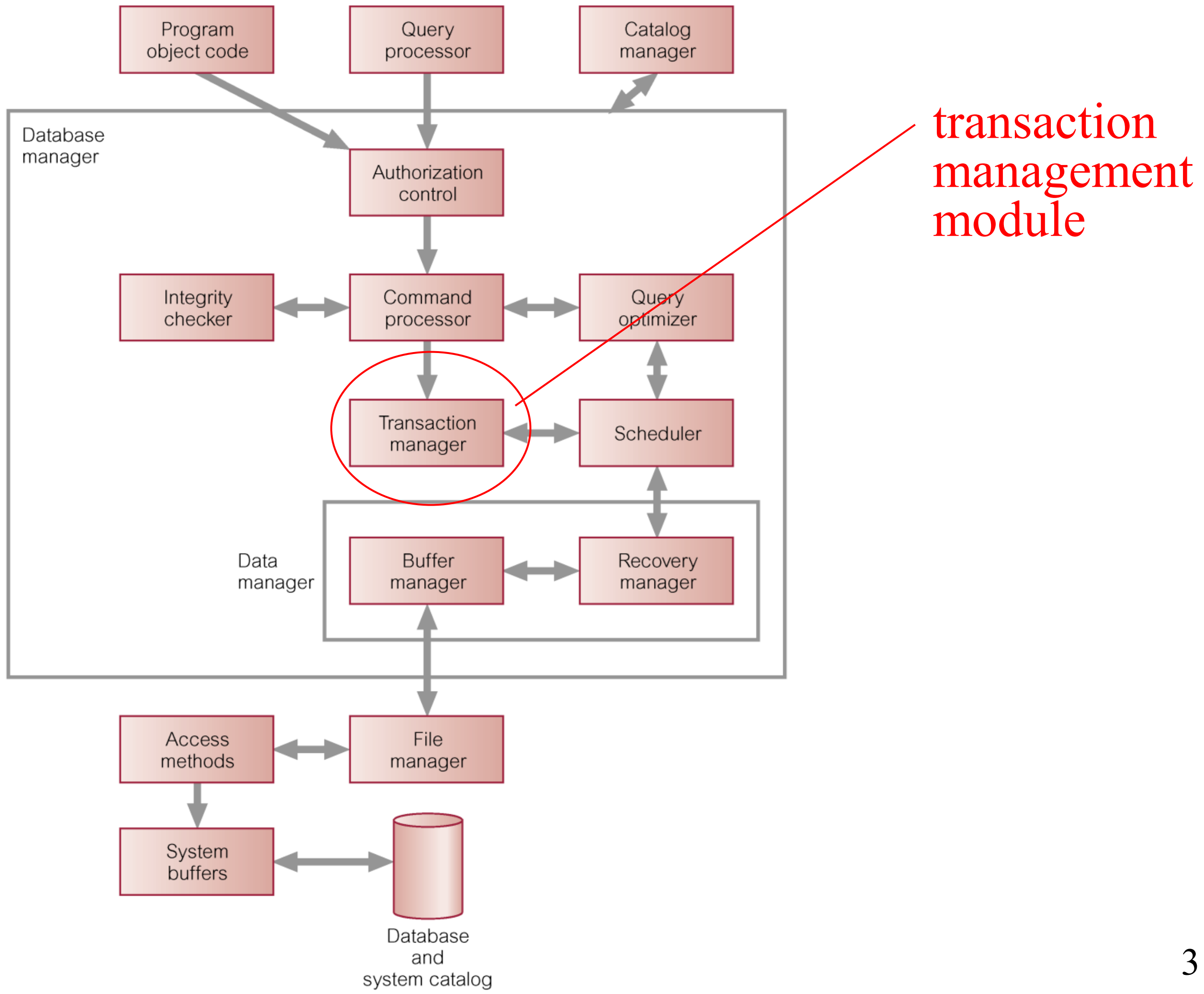
Technologies of Databases: Transaction Management

Architecture of a DBMS: Reliability

A ***transaction*** is a sequence of read and write actions on the database and data in temporary memory, executed ensuring that the DBMS respects the ***ACID*** *properties*

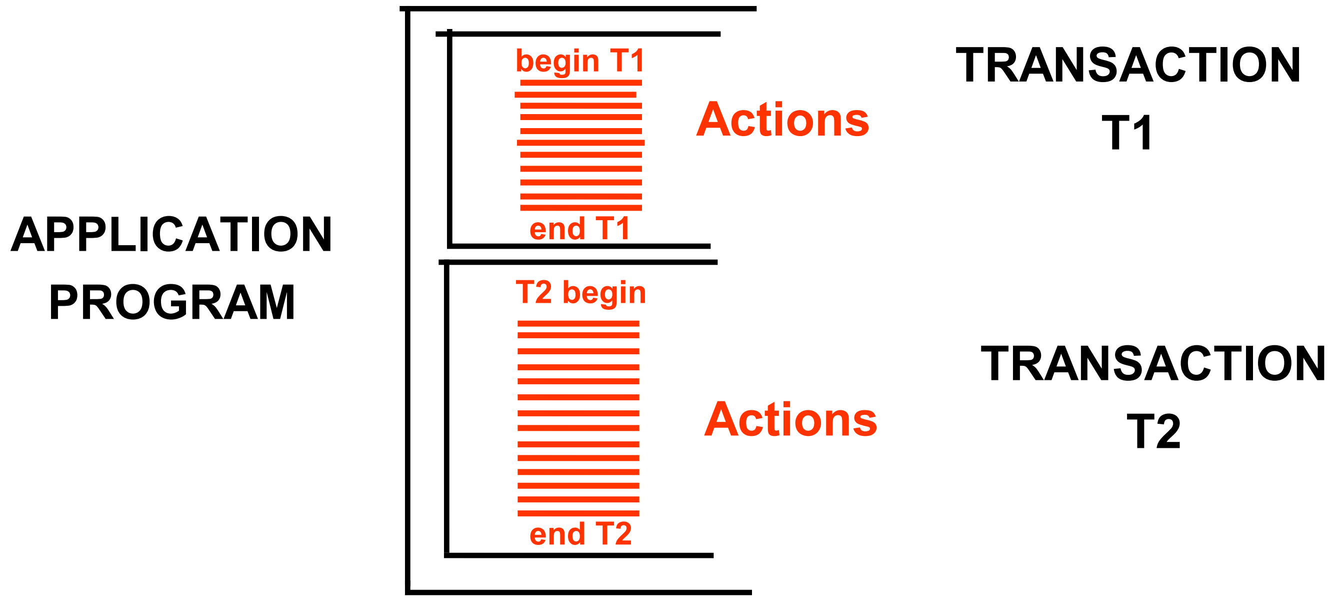
The component modules of a DBMS



Transaction Definition

- Transaction: part of program characterized by a beginning (**begin-transaction**, **`start transaction`** in SQL), an end (**end-transaction,** not explicit in SQL) and inside of which it must be performed one and only one time one of the following commands
  - **commit work** to finish properly
  - **rollback work** to abort the transaction
- A **transactional system (OLTP)** is able to define and execute transactions on behalf of a number of concurrent applications

Difference between application and transaction



A transaction

```
start transaction;
update BankAccount
 set balance = balance + 10
 where AccountNum = 12202;
update BankAccount
  set balance = balance - 10
 where AccountNum = 42177;
commit work;
```

A transaction with various decisions

```
start transaction;
update BankAccount
  set balance = balance + 10 where
  AccountNum = 12202;
update BankAccount
   set balance = balance - 10 where
  AccountNum = 42177;
select Balance into A
from BankAccount
  where AccountNum = 42177;
if (A> = 0) then commit work
     else rollback work;
```

Transactions in JDBC

- Selecting the transaction mode: a method defined in the interface **`Connection`**:

  `setAutoCommit (boolean autoCommit)`
- **`con.setAutoCommit(true)`**
  - (Default) "autocommit": each operation is a transaction
- **`con.setAutoCommit(false)`**
  - the planned transaction management

    `con.commit()`
    `con.rollback()`
  - there is no **`start transaction`**

Architecture of a DBMS: Reliability

Atomicity: A transaction is performed entirety (*committed transaction*) or not performed at all. The transactions that terminate prematurely (*aborted transaction*) are treated as if they had never started.

Consistency preservation: A correct execution of the transaction must bring the DB from one consistent state to another (the integrity constraints must be respected).

Isolation A transaction should not make updates visible to other transactions until it ends normally.

Durability (*persistence*): The changes on the DB of a completed transaction normally are permanent, i.e. they are not alterable by any malfunctions subsequent to termination.

Atomicity

The atomicity has significant operational implications

Outcome

- Commit = "Normal" and more frequent case
  - It can also lead to *remake* the work done to ensure concurrency
- Abort (or rollback)
  - required by the application → suicide
  - required by the system (violation of constraints, concurrency, uncertainty in case of failure) → kill

Consistency

- The transaction must necessarily comply with the integrity constraints
- In other words:
  - if the initial state is correct
  - Also the final state must be correct

If the constraint verification is carried out in mode "*immediate*", it is performed during the transaction, if the verification is carried out in mode "*deferred*", it is carried out during the execution of the commit command.

Isolation

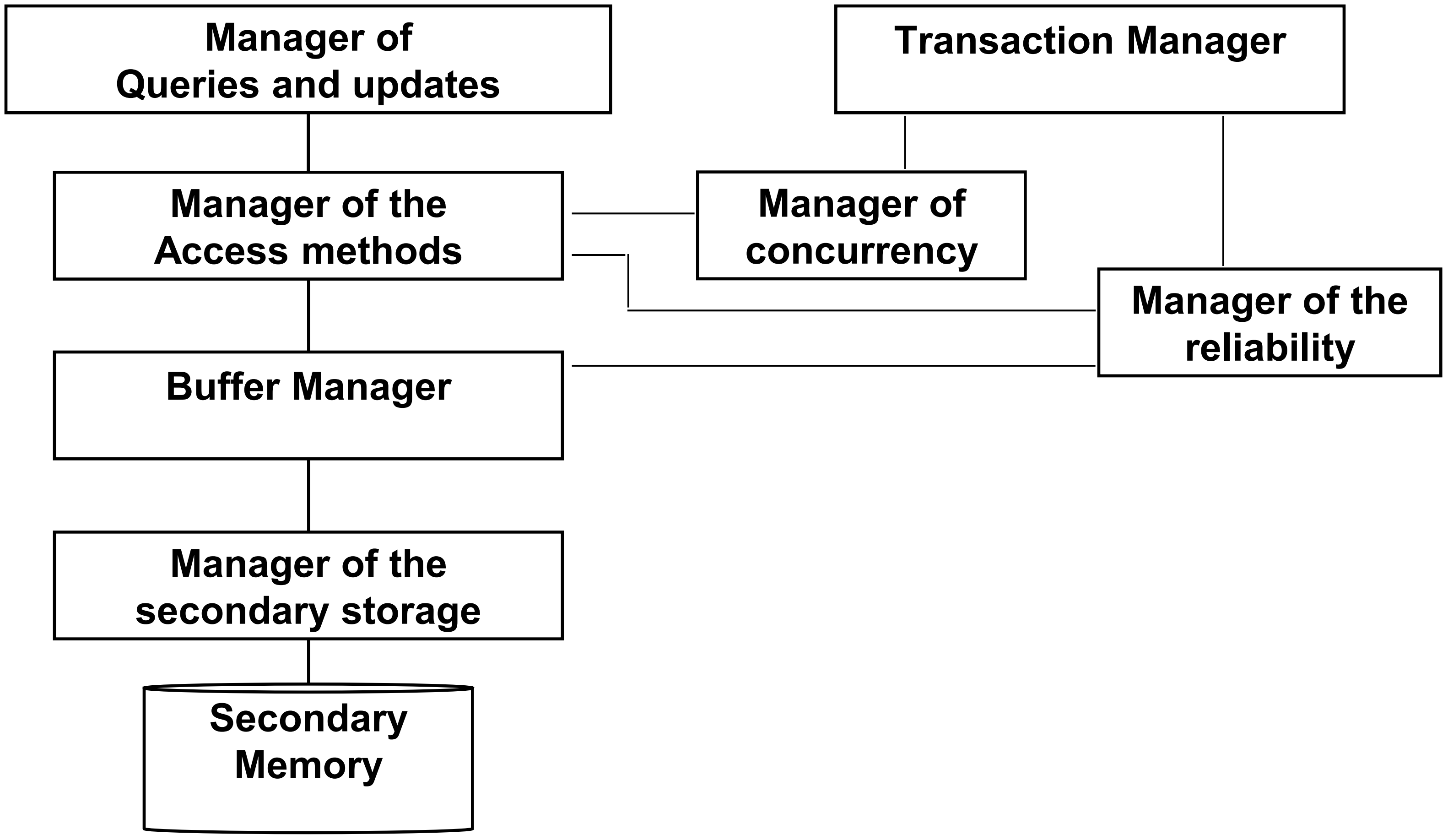
- The transaction is not affected by the effects of other concurrent transactions
  - concurrent execution of a set of transactions must produce a result that could be obtained with a sequential execution
- Consequence: a transaction does not expose its intermediate states
  - Also to avoid the ”domino effect” after rollback

Durability (persistence)

- The effects of a transaction after commit is not lost ("they last forever"), even in the presence of faults
  - Commit means (literally) “commitment”

Access and Query Manager

Transaction Manager


Manager of Queries and updates
Manager of the Access methods
Buffer Manager
Manager of the secondary storage
Secondary Memory
Transaction Manager
Manager of concurrency
Manager of the reliability

Transactions and DBMS modules

- Atomicity and durability
  - Reliability Manager. This module also manages the transfer of data from the buffer to the secondary memory. This is a critical task for the maintenance of durability
- Isolation:
  - Manager of concurrency
- Consistency:
  - integrity manager at runtime, scheduler, (with the support of the DDL compiler)

Coordination is provided by the transaction manager module

Manager of the reliability

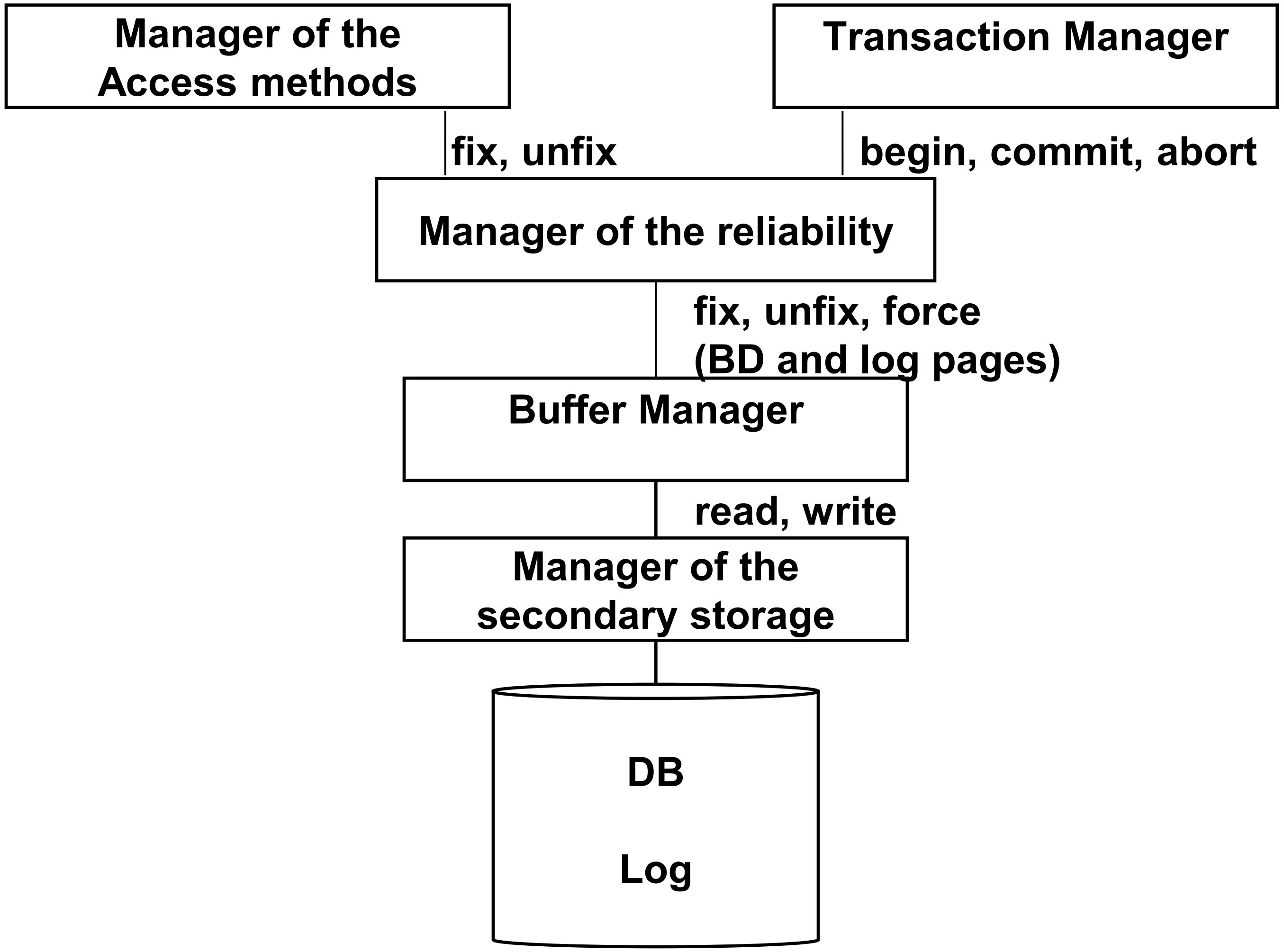
- Ensures atomicity and durability
- Uses the **log**:
  - A permanent archive that records the operations carried out
  - The log can be used for the operations *undo* and *redo*
- It manages the execution of the transactional commands
  - `start transaction` (B, begin)
  - `commit work` (C)
  - `rollback work` (A, abort)

  and restore operations (recovery) after the failure:
  - *warm restart* and *cold* restart

Manager of the reliability

- It receives read and write requests that "passes" to the buffer manager. In addition, it generates its read and write requests that are used to ensure robustness to failures
- To check the reliability, it prepares the data necessary for the execution of recovery operations at failures:
  - *checkpoint* and
  - *dump*

Architecture the manager of reliabilty


Manager of the Access methods
Transaction Manager
fix, unfix
begin, commit, abort
Manager of the reliability
fix, unfix, force (BD and log pages)
Buffer Manager
read, write
Manager of the secondary storage
DB
Log

Stable Memory

In order to operate, the manager of reliability must have a:

- **Stable Memory**, memory that can not be damaged (it is an abstraction):
  - pursued through redundancy:
    - replicated disks
    - strong writing protocols

A typical example of a permanent memory, uses several disk drives ("*mirrored*")

The log files

- The log file is a sequential file which is managed by the reliability manager and is assumed to be stored in a “stable memory” (abstraction)
- It saves all the DDL and DML operations according to the order they are performed
- The log file has a current block (called *top* or, in ORACLE, redo log buffer) where records for single log data are stored. When there is no more empty space in *top*, a new block is created and allocated. This new block is now the *top* block.

The log

- Records in the log
  - *Transaction operations*
    - begin, B(T)
    - insert, I(T,O,AS)
    - delete, D(T,O,BS)
    - update, U(T,O,BS,AS)
    - commit, C(T), abort, A(T)

    These log allow the DBMS to perform *undo* and *redo* operations
  - *System records*
    - `dump`
    - `checkpoint`

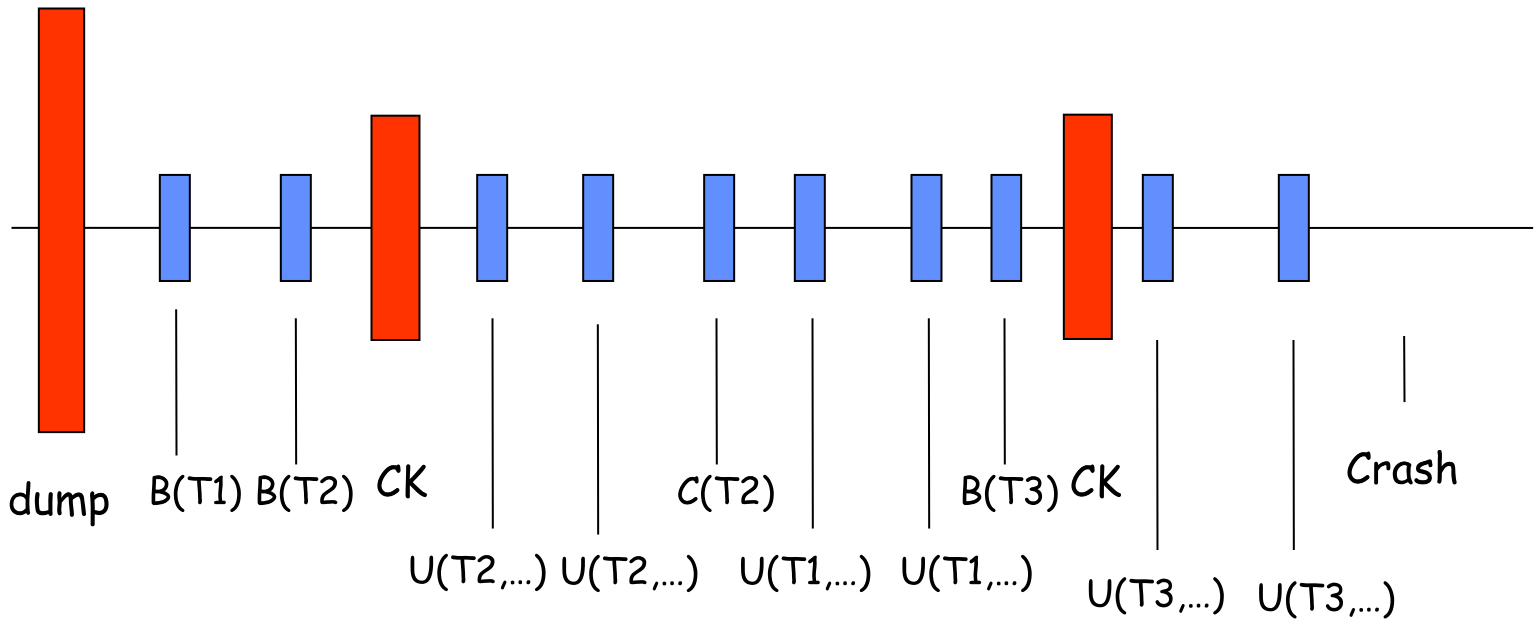
Transaction

Object involved in the operation

Previous value

Current value

The log structure



Log, checkpoint and dump: what are they used for?

- The log is used to “rebuild” operations
- Checkpoint and dump are used to identify the (temporal) point where rebuilding has to start from

Checkpoint

- Checkpoint is an operation which is used to simplify future/possible recovery operations
  - It stores the ID of transactions which are active at a certain point and, thus, confirms that others are either completed or still not started.

Undo and Redo

- Undo of an operation on an object *O:*
  - `update`, `delete`: copy the before state (*BS*) in *O*
  - `insert`: remove *O*

Redo of an operation on an object *O:*
  - `insert`, `update`: copy the after state (*AS*) in *O*
  - `delete`: remove *O*

- Idempotence of *undo* and *redo:*
  - $undo(undo(A)) = undo(A)$
  - $redo(redo(A)) = redo(A)$

Checkpoint

- Simplified algorithm:
  - All requests (of any type) are suspended (writing, insert, …, commit, abort)
  - All the dirty pages of the committed transactions are written on the disk (by means of a *force* operation)
  - The identifiers of all the running transactions are saved in the log (synchronous *force*)
  - All suspended requests are resumed

Checkpoint

- In this way:
  - All the data modified by committed transactions are on the disk
  - Started, but not completed, transactions are listed in the log (checkpoint record)

Dump

- A dump is a complete copy of the database (backup)

  - Usually generated while the DBMS is not working

  - a `dump` record in the log indicates the time-point when the dump has been performed (and additional details such as the file, the URI)

When a transaction is completed

The effect of a transaction is determined when the **commit** record is synchronously written in the log

- A generic failure before such point generates an undo of all the operations in the transaction
- A generic failure after such point should not have consequences: the final state of the database has to be reconstructed, with redo, if necessary

`abort` records can be written asynchronously

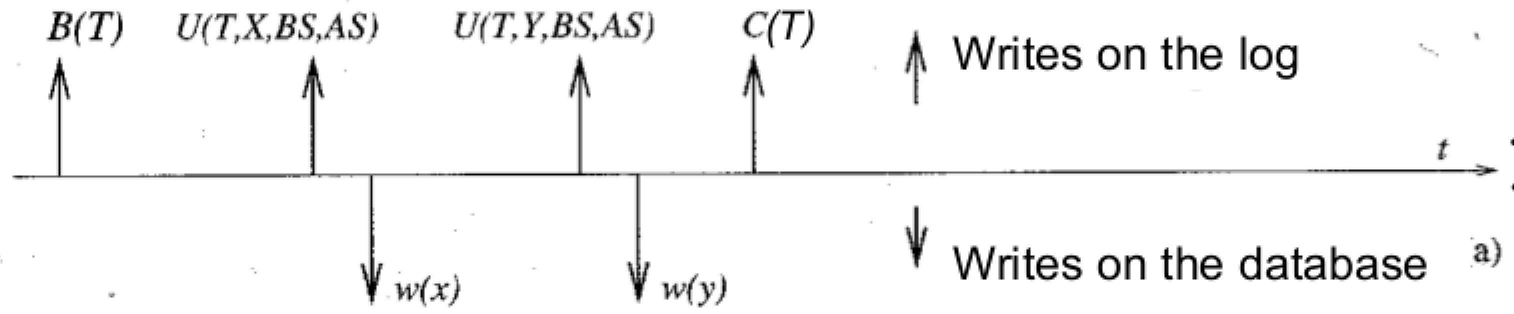
Fundamental rules for the log

- **Write-Ahead-Log (WAL):** (used in ORACLE, all versions)
  - The log records are written in the log *before the corresponding records are written in the secondary storage*
  - This is important for the effectiveness of the Undo operation, because the old value can always be written back to the secondary storage by using the BS value written in the log. In other words, WAL allows the DBMS to undo write operations executed by uncommitted transactions

Fundamental rules for the log

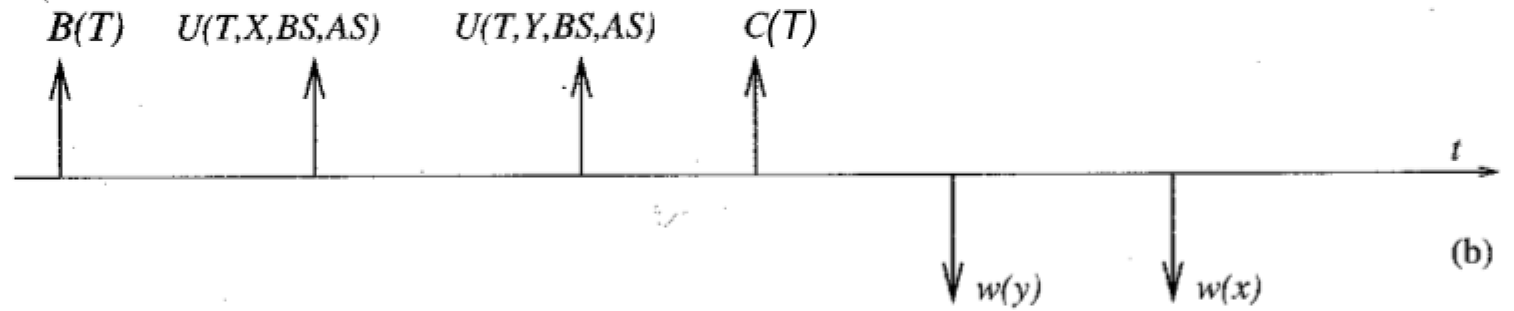
- **Commit-Precedence:**
  - The log records are written in the log before *the commit of the transaction* (and therefore before writing the commit record of the transaction in the log)
  - This is important for the effectiveness of the Redo operation, because if a transaction has been committed before a failure, but its pages have not been written yet in secondary storage, we can use the AS value in the log to write such pages. In other words, the commit-precedence rule allows committed transactions whose effects have not been registered yet in the database to be "re-done"

When to write in the DB?

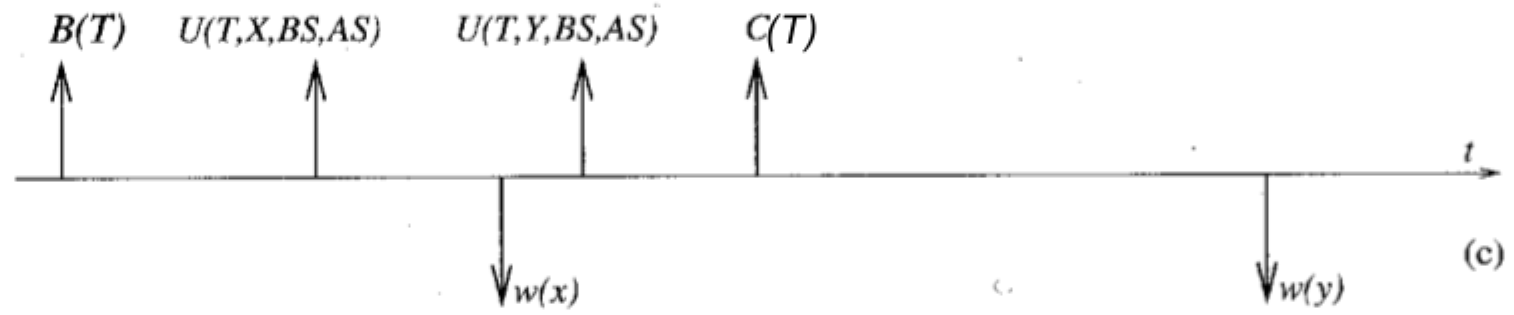


Immediate

All DBMSs



Delayed



Mixed (*)

*Oracle (optionally)

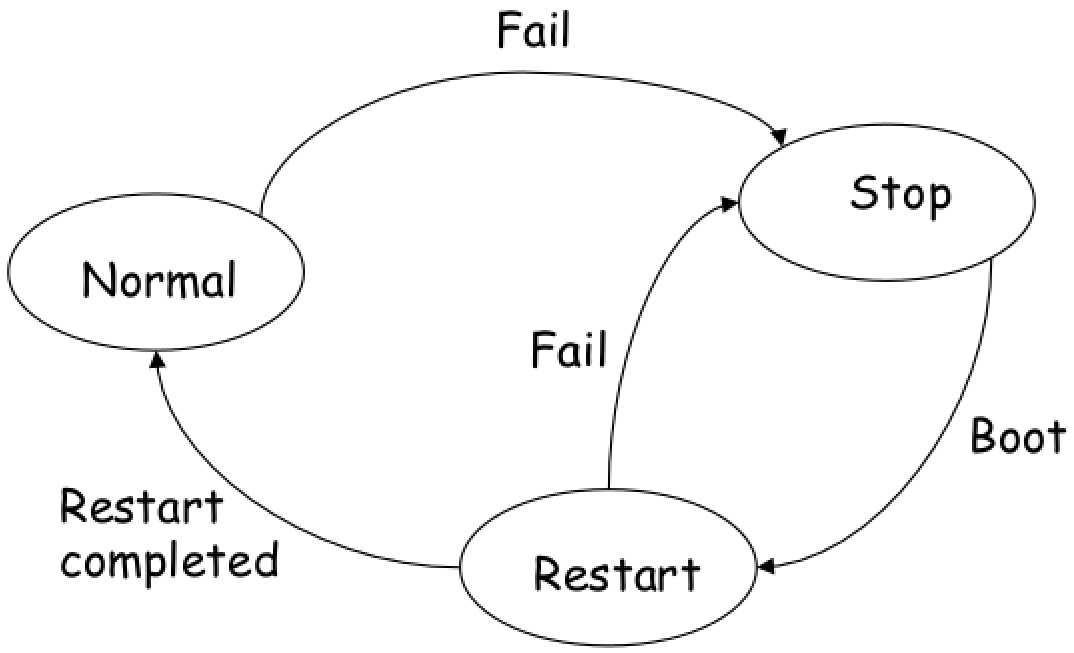
Failures

System failures → warm restart

- **System crash:**
    - **We lose the buffer content, not the secondary storage content**
- **System error or application exception**
    - **E.g. division by zero**
- **Local error conditions of a transaction**
- **Concurrency control**
    - **The scheduler forces the rollback of a transaction**
    - **Storage media failures**

- **Disk failures→ cold restart**
    - **We lose secondary storage content, but not the log file content**
- **"Catastrophic events"→ cold restart???**
    - **Fire**
    - **Flooding**
    - **Etc...**

Failure recovery: the fail-stop model



Restart

- Goal: classify transactions in:
  - Completed (all the data are in the secondary storage)
  - After commit, but not necessarily data are in the secondary storage (a redo can be necessary)
  - Before commit (rollback is necessary, undo)

Warm restart

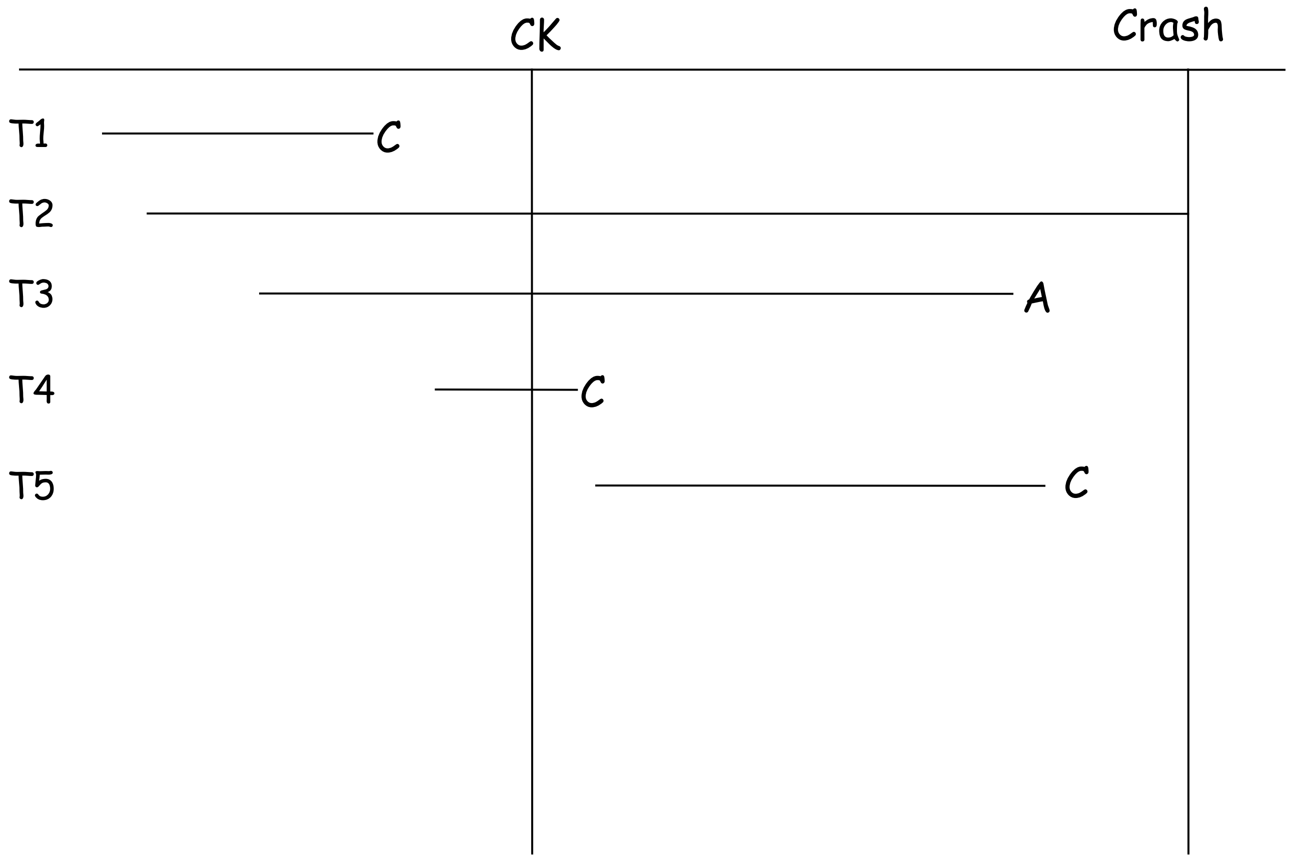
In the mixed effect strategy. The warm restart is composed of 5 steps:

1. Go backward through the log until the most recent checkpoint record in the log
2. Set UNDO={active transactions at checkpoint} REDO={ }
3. Go forward through the log adding to UNDO the transactions with the corresponding begin record, and moving those with the commit record to REDO
4. Undo phase: go backward through the log again, undoing the transactions in UNDO until the begin record of the oldest transaction in the set of active transactions at the last checkpoint (note that we may even go before the most recent checkpoint record)
5. Redo phase: go forward through the log again, redoing the transactions in REDO

Example

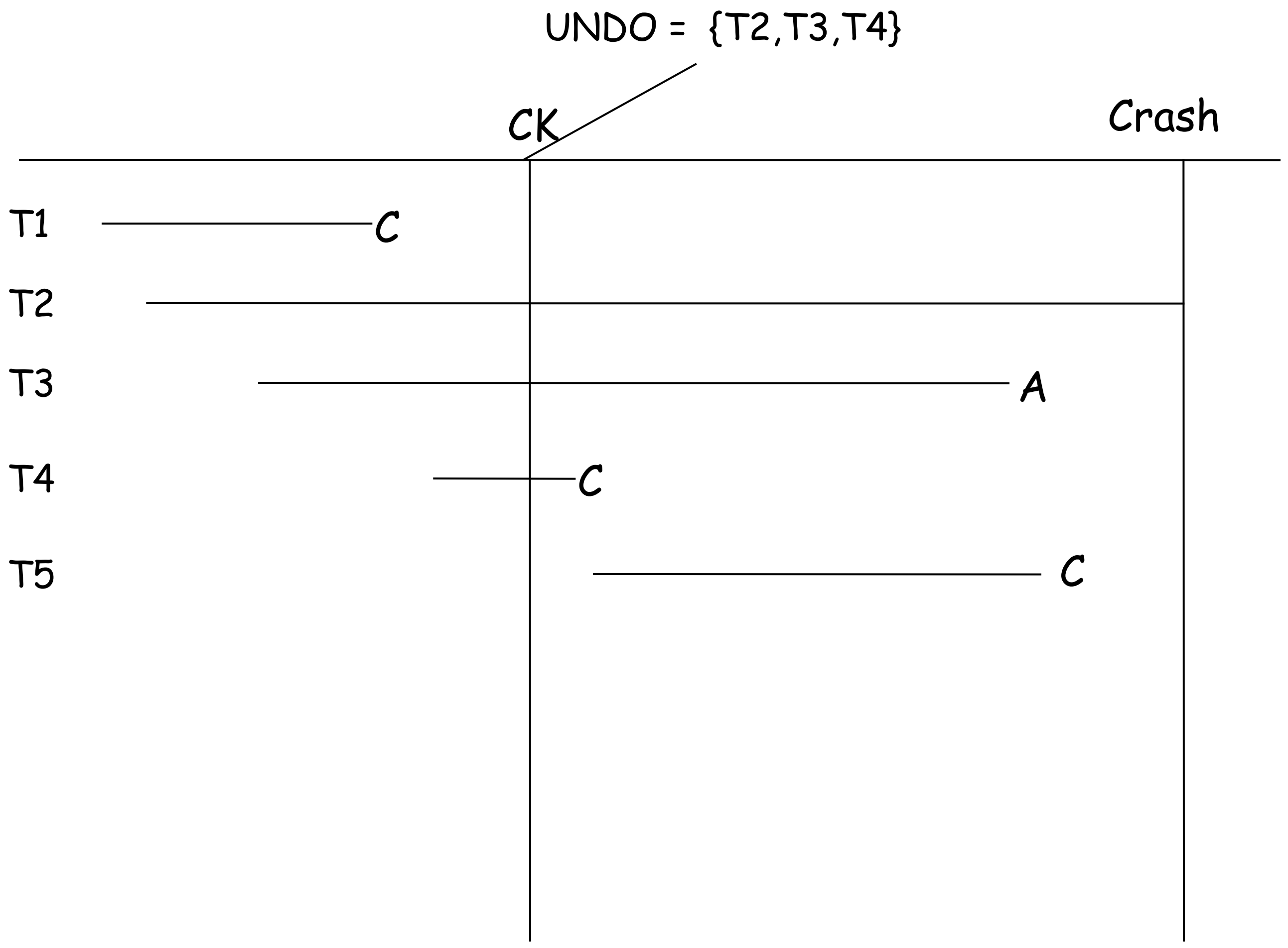
B(T1)
B(T2)
U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
U(T3,O2,B3,A3)
U(T4,O3,B4,A4)
CK(T2,T3,T4)
C(T4)
B(T5)
U(T3,O3,B5,A5)
U(T5,O4,B6,A6)
D(T3,O5,B7)
A(T3)
C(T5)
I(t2,O6,A8)

t



1. Finding the last checkpoint

B(T1)
B(T2)
U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
U(T3,O2,B3,A3)
U(T4,O3,B4,A4)
CK(T2,T3,T4)
C(T4)
B(T5)
U(T3,O3,B5,A5)
U(T5,O4,B6,A6)
D(T3,O5,B7)
A(T3)
C(T5)
I(t2,O6,A8)



2. Building UNDO and REDO

B(T1)
B(T2)
U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
U(T3,O2,B3,A3)
U(T4,O3,B4,A4)
CK(T2,T3,T4)
1. C(T4)
2. B(T5)
U(T3,O3,B5,A5)
U(T5,O4,B6,A6)
D(T3,O5,B7)
A(T3)
3. C(T5)
I(T2,O6,A8)

0. UNDO = {T2,T3,T4}. REDO = {}

---

1. C(T4) $\rightarrow$ UNDO = {T2, T3}. REDO = {T4}

2. B(T5) $\rightarrow$ UNDO = {T2,T3,T5}. REDO = {T4}

3. C(T5) $\rightarrow$ UNDO = {T2,T3}. REDO = {T4, T5}

Setup

3. UNDO phase

B(T1)
B(T2)
8. U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
7. U(T3,O2,B3,A3)
U(T4,O3,B4,A4)
CK(T2,T3,T4)
1. C(T4)
2. B(T5)
6. U(T3,O3,B5,A5)
. U(T5,O4,B6,A6)
5. D(T3,O5,B7)
A(T3)
3. C(T5)
4. I(T2,O6,A8)

0. UNDO = {T2,T3,T4}. REDO = {}

---

1. C(T4) → UNDO = {T2, T3}. REDO = {T4}

2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4}

3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}

Setup

---

4. D(O6)

5. Re-Insert O5 =B7

6. O3 = B5

7. O2 =B3

8. O1=B1

Undo

4. REDO phase

B(T1)
B(T2)
8. U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
C(T1)
B(T4)
7. U(T3,O2,B3,A3)
9. U(T4,O3,B4,A4)
CK(T2,T3,T4)
1. C(T4)
2. B(T5)
6. U(T3,O3,B5,A5)
10. U(T5,O4,B6,A6)
5. D(T3,O5,B7)
A(T3)
3. C(T5)
4. I(T2,O6,A8)

0. UNDO = {T2,T3,T4}. REDO = {}

Setup

1. C(T4) → UNDO = {T2, T3}. REDO = {T4}
2. B(T5) → UNDO = {T2,T3,T5}. REDO = {T4}
3. C(T5) → UNDO = {T2,T3}. REDO = {T4, T5}

Undo

4. D(O6)
5. O5 =B7
6. O3 = B5
7. O2 =B3
8. O1=B1

Redo

9. O3 = A4
10. O4 = A6

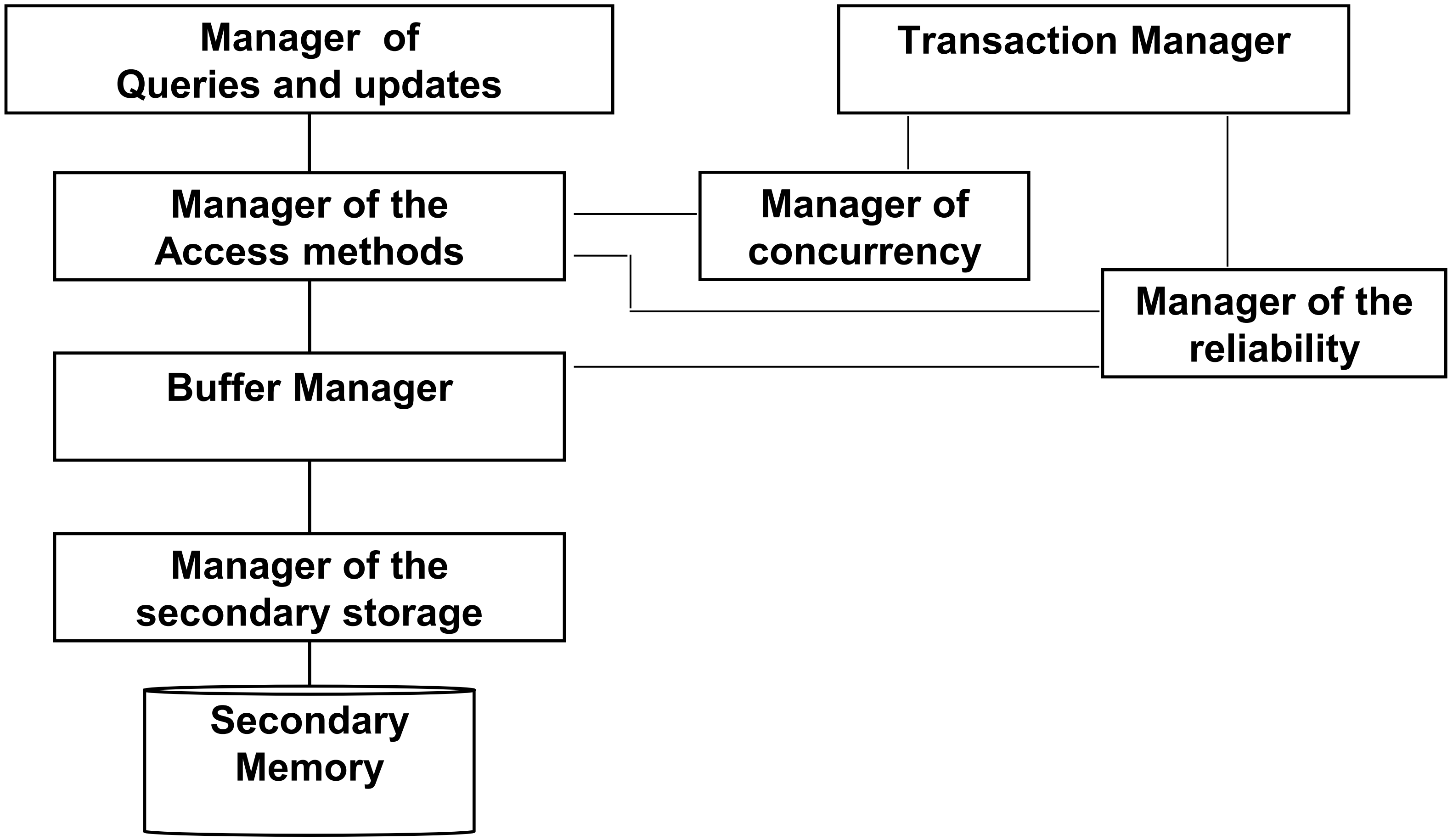
Cold Restart

Three phases:

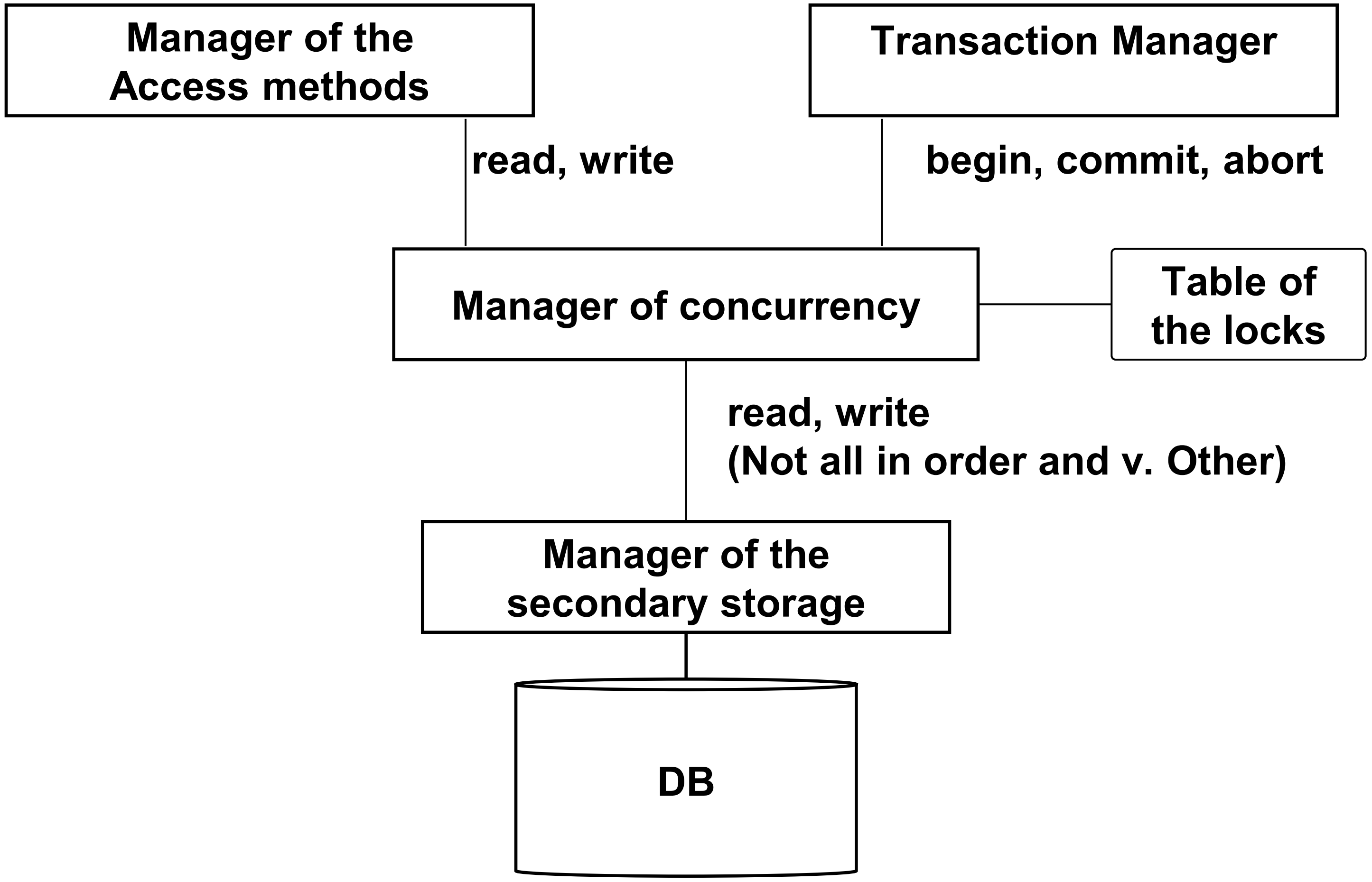
1. Search for the most recent DUMP record in the log, and load the dump into the secondary storage (more precisely, only a selectively copy of the fragments of the DB that have been damaged by the disk failure)
2. Forward recovery of the dump state:
   - Re-apply all actions in the log, in the order determined by the log
   - At this point, we have the database state immediately before the crash
3. Warm restart procedure

Access and query Manager

Transaction Manager


Manager of Queries and updates
Manager of the Access methods
Buffer Manager
Manager of the secondary storage
Secondary Memory
Transaction Manager
Manager of concurrency
Manager of the reliability

Manager of concurrency (Ignoring buffer and reliability)



Details about the process monitor: concurrency control

- Concurrency is fundamental: tens or hundreds of transactions per second cannot be serial:
  - A different (non-serial) schedule is necessary!!!
- Examples: Banks, flight booking

Reference model

- I/O operations on abstract objects *x*, *y*, *z*

Problem

- Anomalies are due to the concurrent execution: they have to be prevented or “controlled”

Anomaly 1: Lost Update

- Two identical transactions
  - $t\,1 : r(\mathrm{x}), x = x + 1, w(\mathrm{x})$
  - $t\,2 : r(\mathrm{x}), x = x + 1, w(\mathrm{x})$
- At the beginning x=2; After a serial execution x=4
- A concurrent execution:

t_1	t_2
$r_1(x)$	
$x = x + 1$	
	$r_2(x)$
	$x = x + 1$
$w_1(x)$	
`commit`	
	$w_2(x)$
	`commit`

- An update is loss: x=3

Anomaly 2: Dirty read

t_1	t_2
$r_1(x)$	
$x = x + 1$	
$w_1(x)$	
	$r_2(x)$
`abort`	
	`commit`

- Critical aspect: t_2 reads an intermediate ("dirty") state and can process it.

Anomaly 3: Non-repeatable Read

- t_1 reads twice

t_1	t_2
`bot`	
$r_1(x)$	
	`bot`
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	`commit`
$r_1(x)$	
`commit`	

- t_1 reads two different values for x !

Anomaly 4: Phantom update

- Let assume that there is the following constraint: $y + z = 1000$;

t_1	t_2
bot	
r_1(y)	
	bot
	$r_2(y)$
	$y = y - 100$
	$r_2(z)$
	$z = z + 100$
	$w_2(y)$
	$w_2(z)$
	commit
$r_1(z)$	
$s = y + z$	
commit	

- $s = 1100$: the constraint seems to be not satisfied, t_1 reads a non-coherent value.

Anomaly 5: Phantom insert

t_1 t_2

"reads salaries of employees of dept. A and computes the average"

"inserts a new a new employee in A"

`commit`

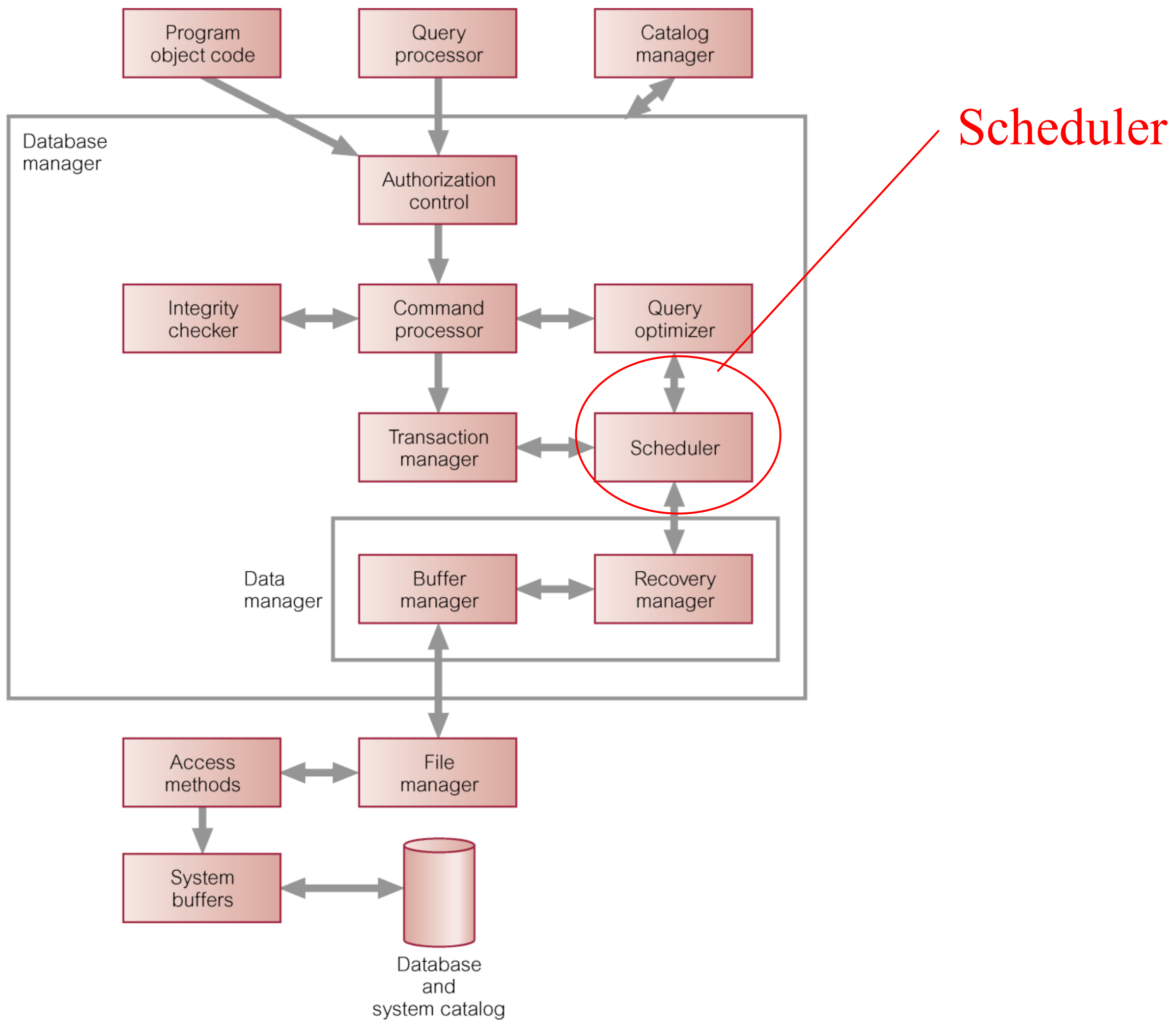
"reads salaries of employees of dept. A and computes the average"

`commit`

Anomalies

• Lost Update	W-W
• Dirty read	R-W (or W-W) with abort
• Non-repeatable read	R-W
• Phantom update	R-W
• Phantom insert	R-W on a new tuple

The component modules of a DBMS



Schedule

- A sequence of input / output operations of concurrent transactions
- Example:

$$S_1 : r_1(x)\ r_2(z)\ w_1(x)\ w_2(z)$$

- Simplifying assumptions (which will remove in the future, as not acceptable in practice):
  - consider **commit-projection**: we ignore transactions that go into abort, removing all their actions from schedules

Concurrency control

The goal of the concurrency control phase is to avoid anomalies

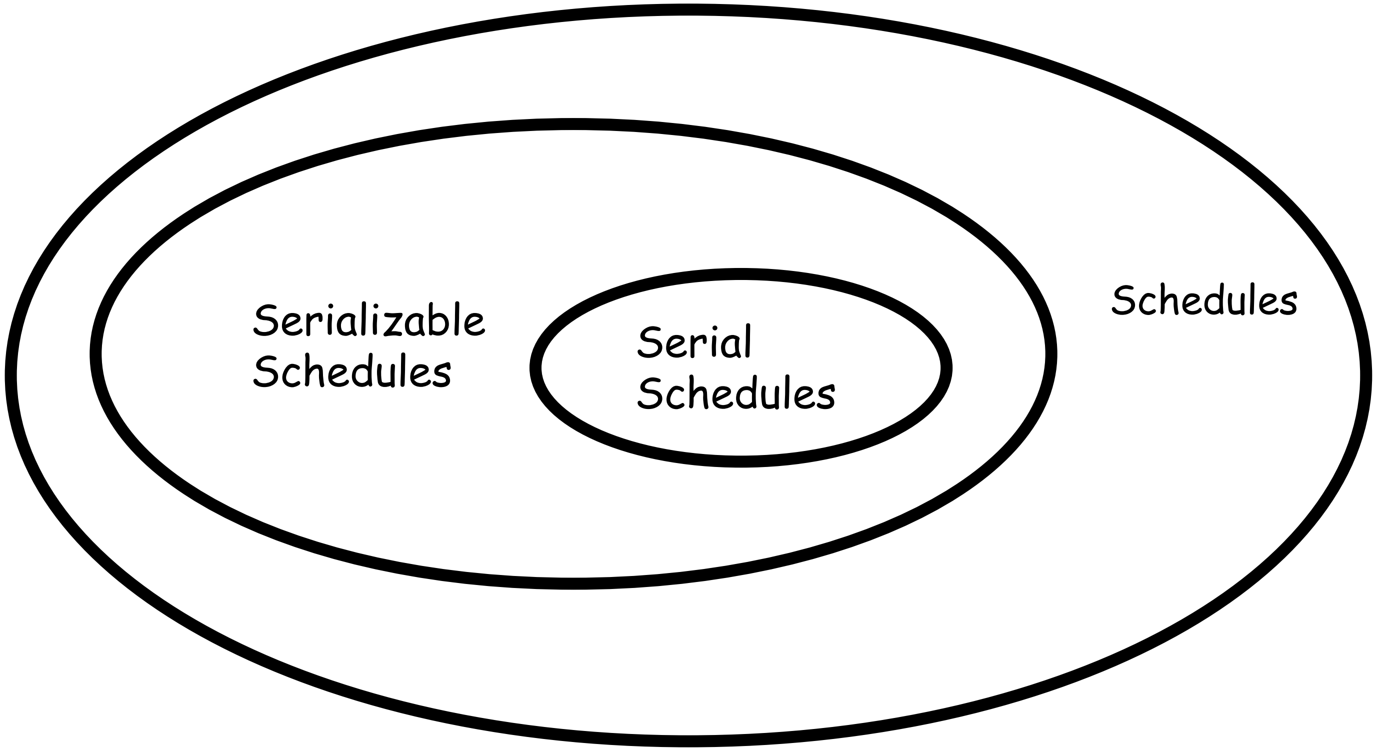
- *Scheduler*: A system that accepts or rejects (or reorders) the operations required by the transactions

Some definitions:

- *Serial Schedule:* The transactions are separated, one at a time $S_2 : r_0(x)\ r_0(y)\ w_0(x)\ r_1(y)\ r_1(x)\ w_1(y)\ r_2(x)\ r_2(y)\ r_2(z)\ w_2(z)$
- *Serializable Schedule:* It produces the same result of a serial schedule on the same transactions
  - It requires a notion of equivalence between schedule

Basic idea

- Define classes of serializable schedules that are subclasses of possible schedules, and whose serializability property is verifiable with cheap algorithm



View-Serializability

- Definitions:
  - $r_i(x)$ ***reads-from*** $w_j(x)$ in a schedule S if $w_j(x)$ precedes $r_i(x)$ in S and there is no $w_k(x)$ between $r_i(x)$ and $w_j(x)$ in S
  - $w_i(x)$ in a schedule S is ***final writing*** *if* it's the last writing of the object x in S
- Two schedules are ***view-equivalent*** ($S_i \approx_V S_j$) if have the same relationships *reads-from* and the same *final writings*
- A schedule is ***view-serializable*** if it is view-equivalent to some serial schedule
- The set of view-serializable schedules is indicated with **VSR**

View-Serializability: Examples

$S_3 : w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$

$S_4 : w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$

View-Serializability: Examples

$S_3 : w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$

$S_4 : w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$

– S_3 is view-equivalent the serial schedule S_4 (and therefore is view-serializable)

View-Serializability: Examples

$S_3 : w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$
$S_4 : w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$
$S_5 : w_0(x)\ r_1(x)\ w_1(x)\ r_2(x)\ w_1(z)$
$S_6 : w_0(x)\ r_1(x)\ w_1(x)\ w_1(z)\ r_2(x)$

– S_3 is view-equivalent the serial schedule S_4 (and therefore is view-serializable)

View-Serializability: Examples

$S_3 : w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$
$S_4 : w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$
$S_5 : w_0(x)\ r_1(x)\ w_1(x)\ r_2(x)\ w_1(z)$
$S_6 : w_0(x)\ r_1(x)\ w_1(x)\ w_1(z)\ r_2(x)$

- S_3 is view-equivalent the serial schedule S_4 (and therefore is view-serializable)
- S_5 is not view-equivalent to S_4but is view-equivalent to the serial schedule S_6, and then is view-serializable

View-Serializability: Examples

$S_7 : r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$ (Update lost)

$S_8 : r_1(x)\ r_2(x)\ w_2(x)\ r_1(x)$ (Non-repeatable reads)

$S_9 : r_1(x)\ r_1(y)\ r_2(z)\ r_2(y)\ w_2(y)\ w_2(z)\ r_1(z)$

(Phantom update)

– S_7, S_8 S_9 are not view-serializable

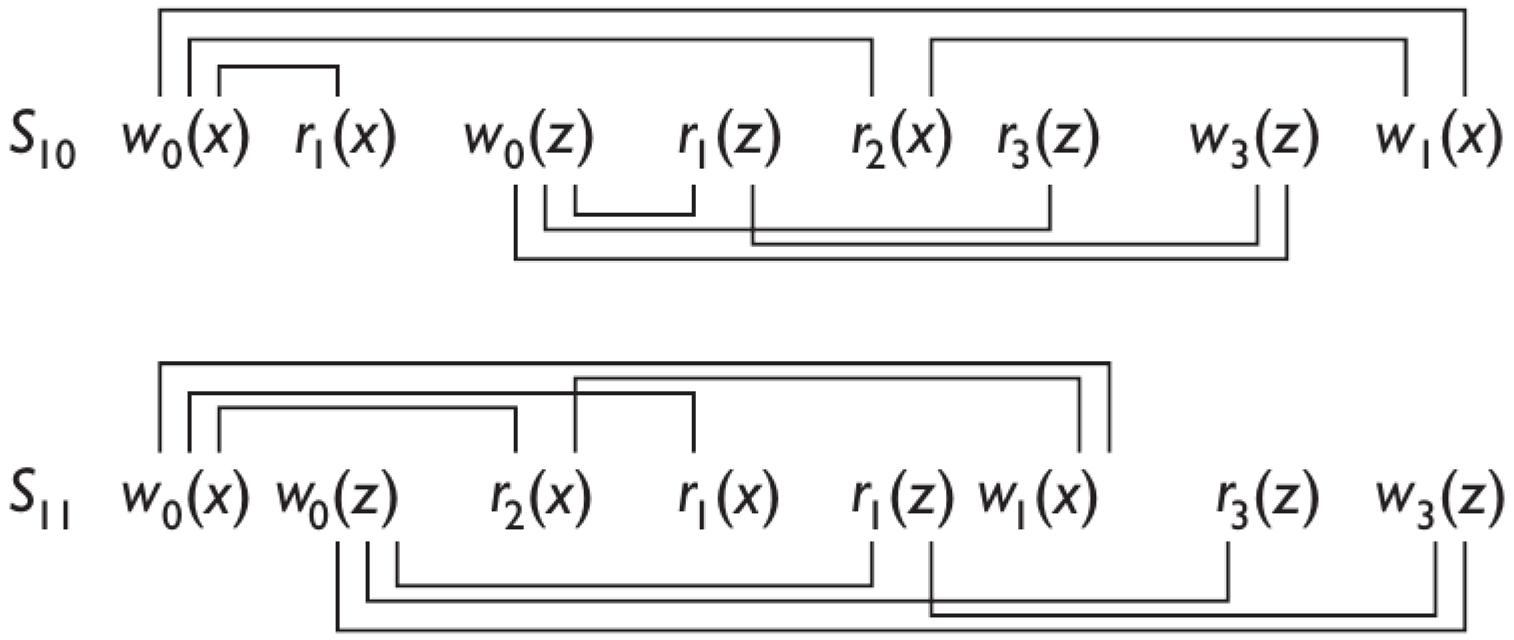
View-serializability

- Complexity:
  - the verification of view-equivalence of two given schedules:
    - Polynomial
  - decide on the View serializability of one schedule:
    - NP-complete problem
- It is not usable in practice

Conflict-serializability

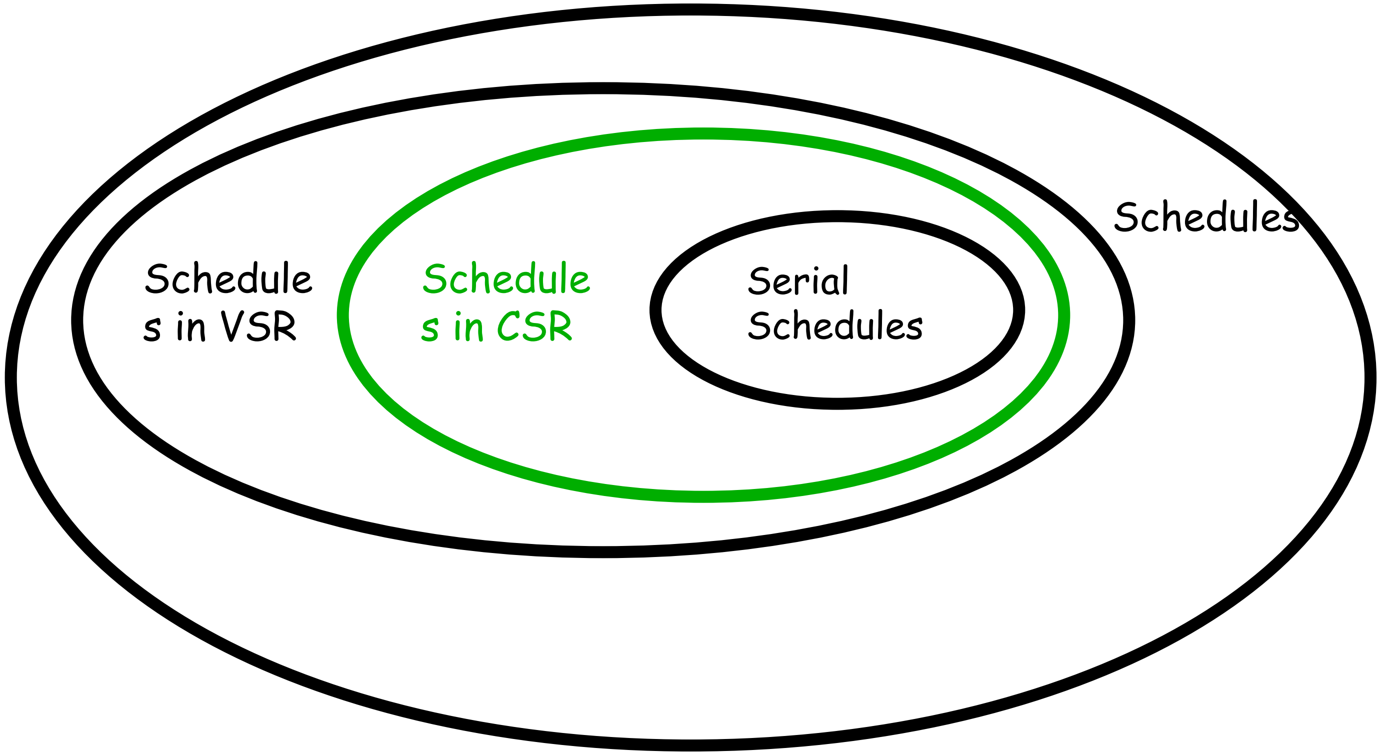
- Preliminary definition:
  - An action a_i is in *conflict* with a_j ($i \neq j$), if they operate on the same object and at least one of them is a write operation. Two cases:
    - conflict *read-write* (*rw* or *wr*)
    - conflict *write-write* (*ww*).
- *Conflict-equivalent schedules* ($S_i \approx_C S_j$): They include the same operations and each pair of conflicting operations appear in the same order in both
- A schedule is *conflict-serializable* if it is conflict-equivalent to some serial schedule
- The set of schedules conflict-serializable is indicated with **CSR**

An example of a schedule conflict-equivalent to a serial schedule


S_{10} $w_0(x)$ $r_1(x)$ $w_0(z)$ $r_1(z)$ $r_2(x)$ $r_3(z)$ $w_3(z)$ $w_1(x)$
S_{11} $w_0(x)$ $w_0(z)$ $r_2(x)$ $r_1(x)$ $r_1(z)$ $w_1(x)$ $r_3(z)$ $w_3(z)$

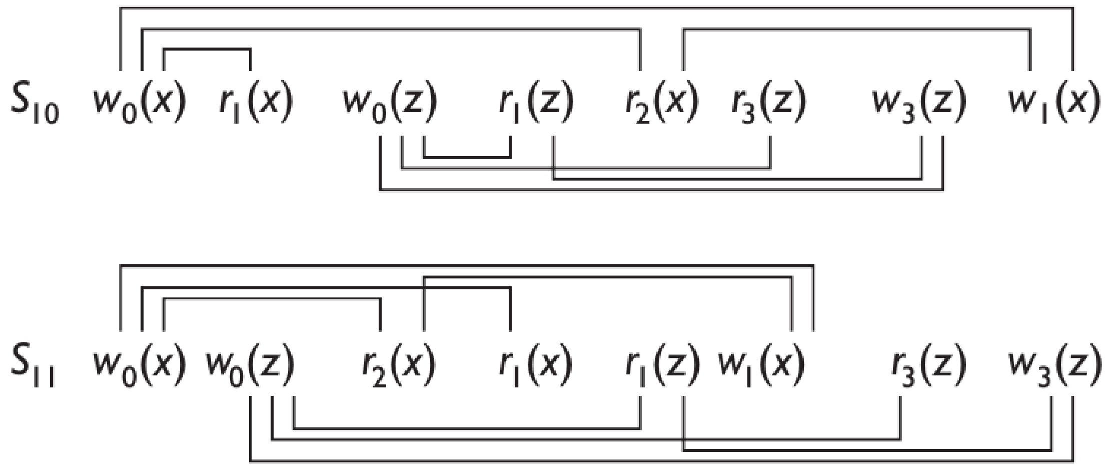
CSR and VSR

It can be shown that the Conflict-serializability implies the view-serializability



Check the conflict-serializability

- Through the **conflict graph**:
  - a node for each transaction t_i
  - an edge (oriented) by t_i to t_j if there is at least one conflict between action a_i and action a_j such that a_i precedes a_j
- Theorem
  - **a schedule it is CSR if and only if the graph it is acyclic**



S_{10} $w_0(x)$ $r_1(x)$ $w_0(z)$ $r_1(z)$ $r_2(x)$ $r_3(z)$ $w_3(z)$ $w_1(x)$
S_{11} $w_0(x)$ $w_0(z)$ $r_2(x)$ $r_1(x)$ $r_1(z)$ $w_1(x)$ $r_3(z)$ $w_3(z)$

$$\begin{tikzcd}
 & t_0 \arrow[dl] \arrow[dr] \arrow[dd] & \\
t_1 \arrow[dr] & & t_2 \arrow[ll] \\
 & t_3 &
\end{tikzcd}$$

Controlling Concurrency in Practice

- The conflict-serializability, even if more rapidly verifiable (the algorithm, with appropriate data structures requires linear time), is unusable in practice
- The technique would be efficient if we could know the graph from the start, but this is not the case: a scheduler must operate "incrementally", that is, to each action immediately decide whether to run or do something else;
  It is not practicable to maintain the graph, update and verify the acyclicity to each transaction request

Controlling Concurrency in Practice

- In addition, the technique is based on the assumption of commit-projection
- In practice, DBMSs use techniques that
  - ensure conflict-serializability without having to build the graph
  - do not require the assumption of commit-projection

Lock

- Basic idea:
  - All the reading operations are performed after an *r_lock* (shared lock) and before an *unlock*
  - All the writing operations are performed after a *w_lock* (exclusive lock) and before an *unlock*
- When a transaction first reads and then writes an object, it can:
  - Ask for an exclusive lock (directly)
  - First ask for a shared lock and then for an exclusive lock (*lock escalation*)
- The *lock manager* receives such requests from the transactions and accepts or refuses them, on the basis of a conflict table

Managing the locks

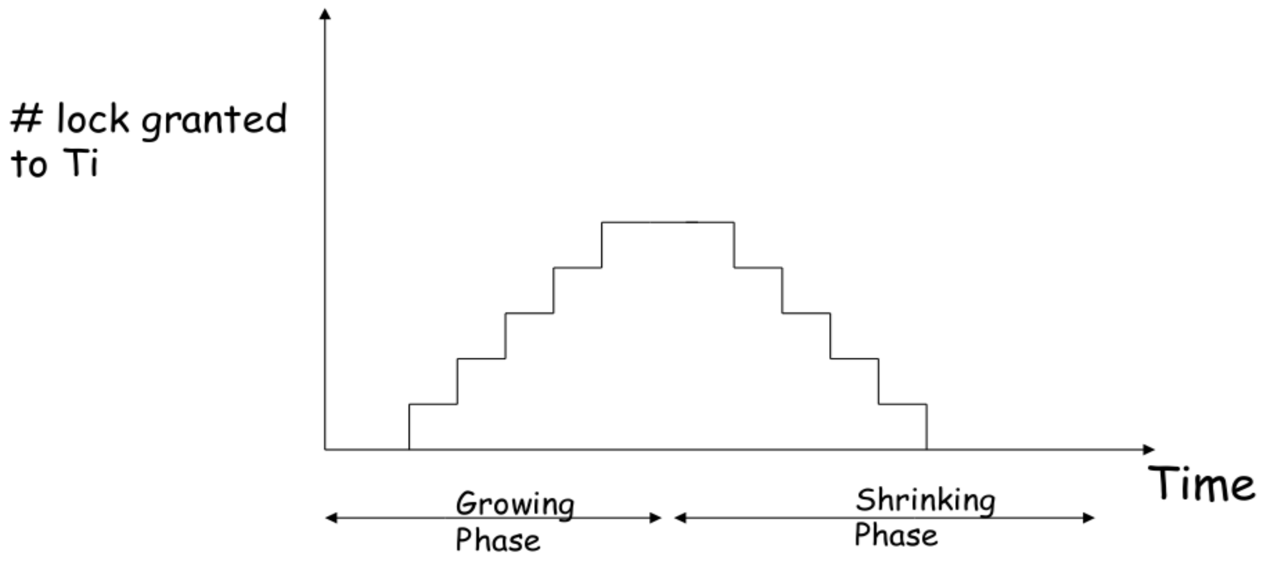
- Conflict table

	Obj status		
Request	**free**	**r_locked**	**w_locked**
r_lock	OK / r_locked	OK / r_locked	NO/ w_locked
w_lock	OK / w_locked	NO / r_locked	NO / w_locked
unlock	error	OK / depends	OK / free

- A counter is used to keep track of the number of “readers”. The object is released when this counter is 0
- If the request is accepted, the transaction is enqueued (wait) till the object becomes available
- The manager of concurrency manages the “lock table”.

Two-Phase Locking (2PL)

- Used by most of the systems (ORACLE included)
- A schedule with exclusive locks follows the two-phase locking protocol if in each transaction Ti appearing in the schedule, all lock operations precede all unlock operations



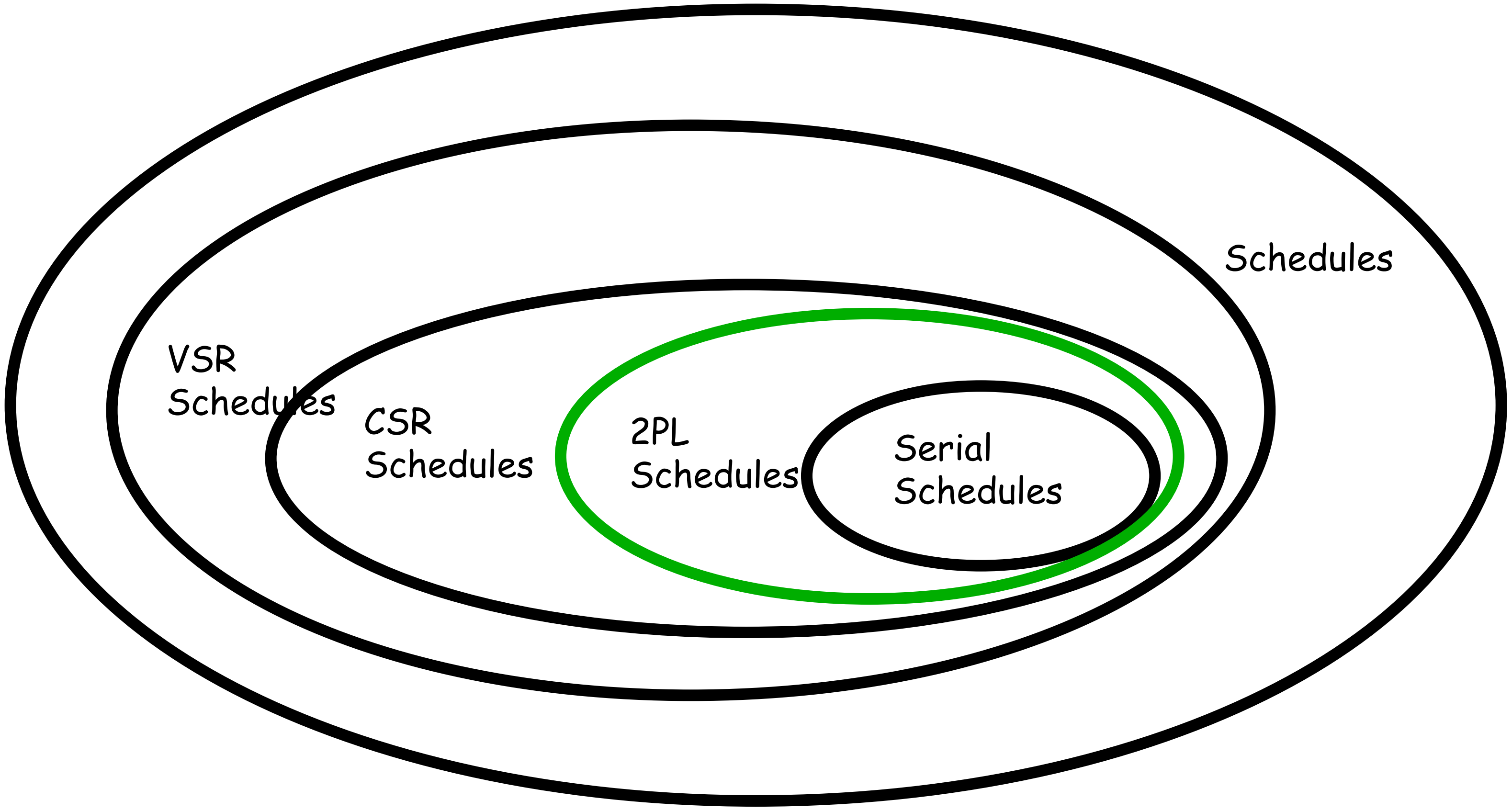
2PL and CSR

- Each schedule 2PL is conflict serializable (the contrary is not necessarily true)
- Counterexample:

  $r_1(X)\ w_1(X)\ r_2(X)\ w_2(X)\ r_3(Y)\ w_1(Y)$

  – Violates 2PL
  – Is conflict-serializable

CSR, VSR and 2PL



Strict *2PL*

- Additional Conditions:
  - **The lock can be released only after commit or abort**
- It overcomes the need of commit-projection (and eliminates the risk of dirty reads)

t_1	t_2	x	y	z
`bot`		free	free	free
$r\_lock_1(x)$		1:read		
$r_1(x)$				
	`bot`			
	$w\_lock_2(y)$		2:write	
	$r_2(y)$			
$r\_lock_1(y)$			1:wait	
	$y = y - 100$			
	$w\_lock_2(z)$			2:write
	$r_2(z)$			
	$z = z + 100$			
	$w_2(y)$			
	$w_2(z)$			
	`commit`			
	$unlock_2(y)$		1:read	
$r_1(y)$				
$r\_lock_1(z)$				1:wait
	$unlock_2(z)$			1:read
$r_1(z)$				
	`eot`			
$s = x + y + z$				
`commit`				
$unlock_1(x)$		free		
$unlock_1(y)$			free	
$unlock_1(z)$				free
`eot`				

Concurrency control based on timestamp

- Alternative to 2PL
- **Timestamp**:
  - identifier that defines a total order of system events
- Every transaction has a timestamp that represents the moment it started
- A schedule is accepted only if it reflects the serial order of the transactions led by timestamp (TS)

Details

- The scheduler has two counters RTM(x) and WTM(x) for each object
- The scheduler receives requests for reads and writes (indicated with the transaction timestamp):
  - *read*(x,*ts*):
    - If $ts <$ WTM(X) then the request is rejected and the transaction is killed;
    - otherwise, the request is accepted and RTM(x) is set equal to the greater between RTM(x) and *ts*
  - *write*(x,*ts*):
    - If $ts <$ WTM(x) or $ts <$ RTM(x) then the request is rejected and the transaction is killed,
    - otherwise, the request is accepted and WTM(x) is set equal to *ts*
  - If there are transactions which depend on a transaction killed, they are killed.

- Many transactions are killed
- To operate without possibility of commit-projection, the DBMS needs to "buffer" the writing operations until commit (with waitings)

Example

RTM (x) = 7

WTM (x) = 4

Request	Answer	New value
read(x, 6)	ok	
read(x, 8)	ok	RTM (x) = 8
read(x, 9)	ok	RTM (x) = 9
write(x, 8)	no,	t_8 killed
write(x, 11)	ok	WTM (x) = 11
read(x, 10)	no, t_{10} killed	

2PL vs TS

- They are incomparable
  - Schedule in TS and CSR but not in 2PL

$r_1(x)\ w_2(x)\ r_3(x)\ r_1(y)\ w_2(y)\ r_1(v)\ w_3(v)\ r_4(v)\ w_4(y)\ w_5(y)$

$$\begin{tikzcd}
t_1 \arrow[r] \arrow[dd] \arrow[ddr] \arrow[drr] & t_2 \arrow[ddl] \arrow[dd] \arrow[dr] & \\
 & & t_5 \\
t_3 \arrow[r] & t_4 \arrow[ur] &
\end{tikzcd}$$

2PL vs TS

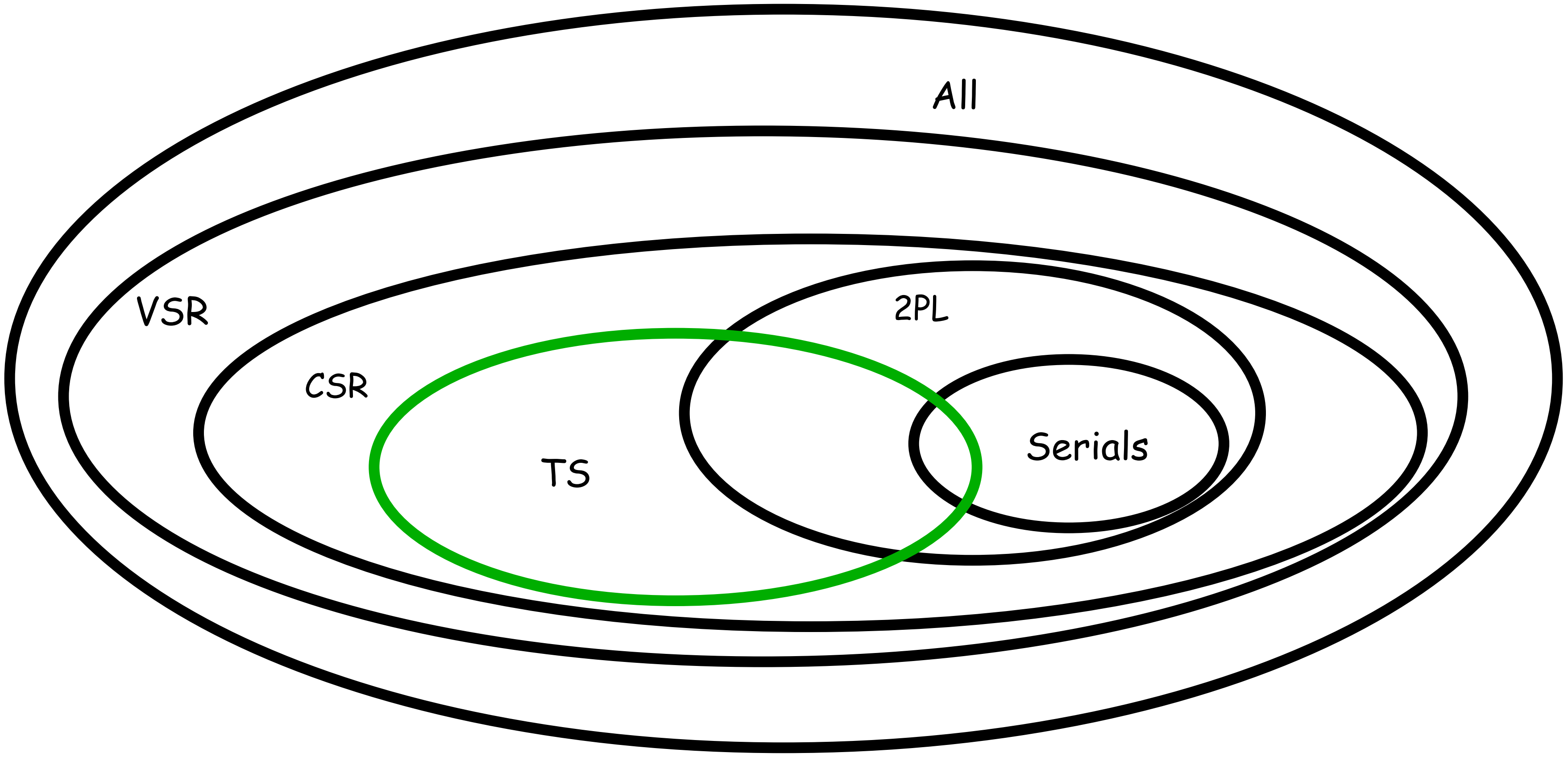
- Schedule in 2PL but not in TS

  $r_2(X)\ w_2(X)\ r_1(X)\ w_1(X)$

- Schedule in TS and 2PL

  $r_1(X)\ r_2(Y)\ w_2(Y)\ w_1(X)\ r_2(X)\ w_2(X)$

CSR, VSR, 2PL and TS



2PL vs TS

- In 2PL transactions are put on hold
- In TS they are killed and rerun
- To remove the commit projection, waiting for commit (both for 2PL and TS)
- 2PL can cause deadlock (we'll see)
- The restarts are usually more costly than waitings:
  - 2PL is more convinient

Mechanisms for the lock management

Locks typically are a little bit more elaborated in their representation than we have seen so far

r_lock (T, x, errcode, timeout)

w_lock (T, x, errcode, timeout)

unlock (T, x)

- *errcode* represents a value returned by the lock manager (0 for satisfied requests, error code otherwise)

Mechanisms for the lock management

- *timeout* is the time value that indicates the timeout for the lock request before it is cancelled. The cancellation may cause a ROLLBACK or a restart of the transaction.

To lock tables are accessed very frequently. For this reason, they are maintained in main memory (buffer).

For each *object* involved in transactions the status of the lock (2 bits) and a counter that represents the number of waiting processes are maintained.

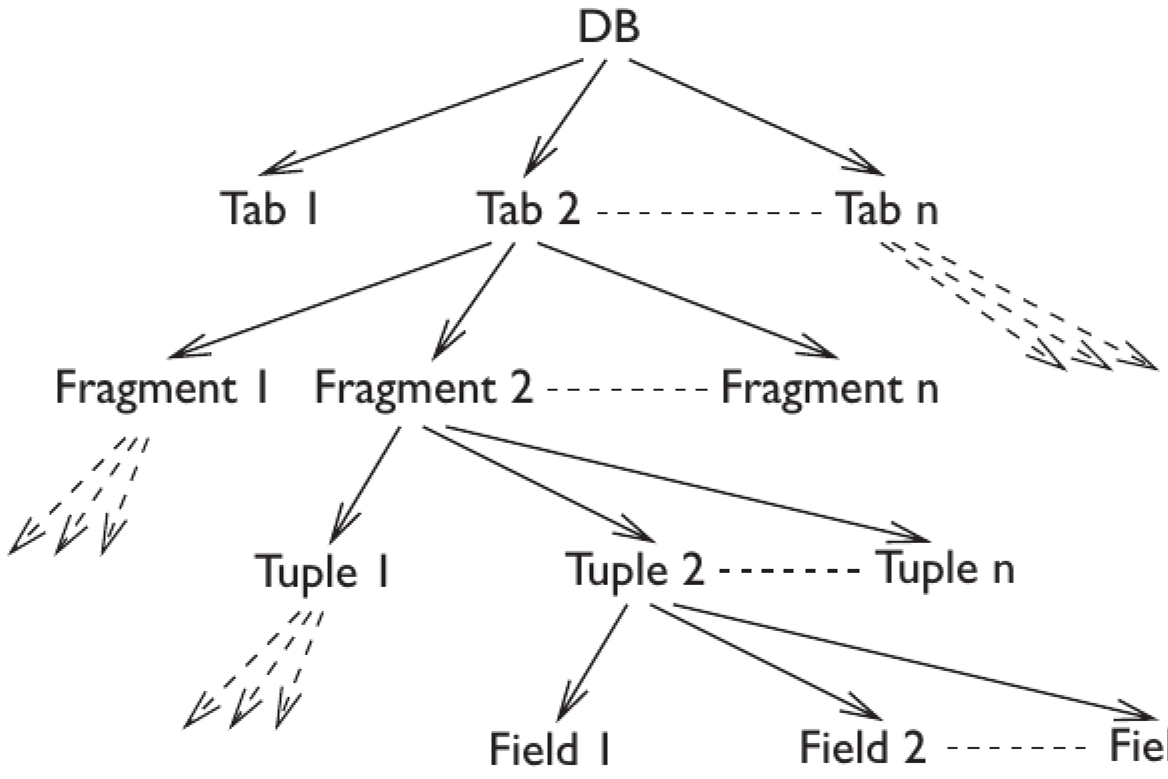
Mechanisms for the lock management (hierarchical lock)

But what are the *objects*? They can be

- The entire Database
- Tables
- Partition Tables
- Tuples
- (More rarely) single tuple fields

The DBMS take into account the different types of objects involved in lock operations and use hierarchical management mechanisms.

Mechanisms for the lock management (hierarchical lock)



Mechanisms for the lock management (hierarchical lock)

5 primitives requested lock:

- XL (exclusive lock). Write lock of the non-hierarchical protocol
- SL (shared lock). Read lock of the non-hierarchical protocol
- ISL (intention shared lock). It expresses the intention of shared lock of one of the nodes under the current node
- IXL (intention exclusive lock). It expresses the intention of exclusive lock of one of the nodes under the current node
- SIXL (shared intention-exclusive lock). It locks the current node in a shared mode and expresses the intention of exclusively locking one of the nodes descending from the current node.

Mechanisms for the lock management (hierarchical lock)

Example

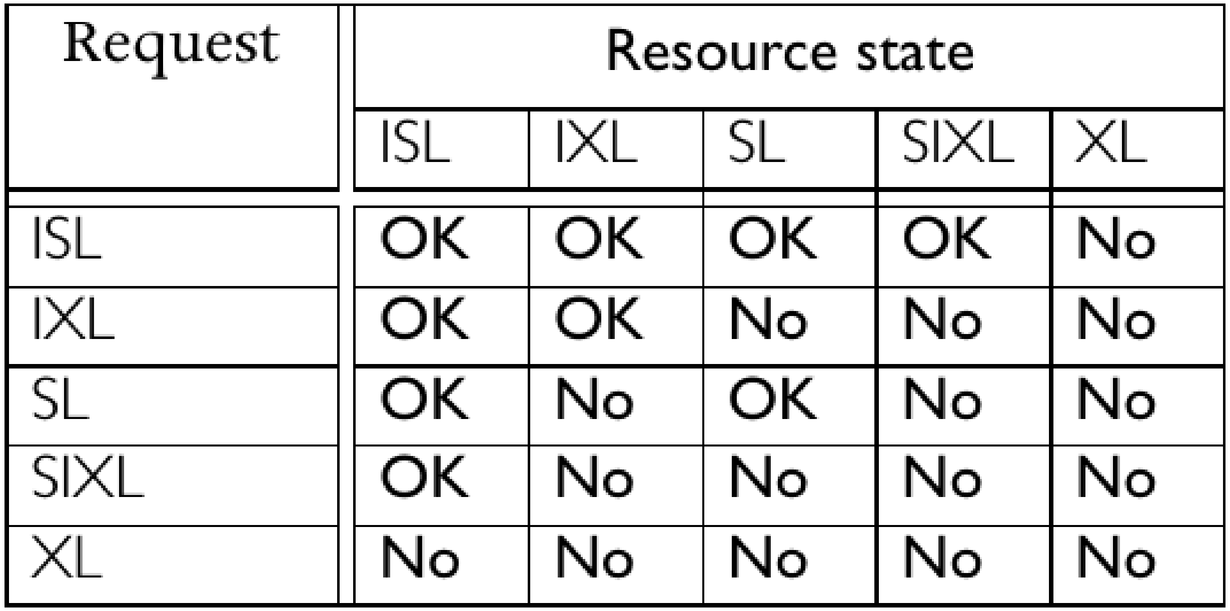
If we wanted to lock in writing a tuple of the table, then we must first request a IXL at the level of the data base. When the request is satisfied, we can ask in a sequence IXL for the table and for the partition that forms the table. Then we will ask for a lock on tuple XL and after the execution of writing, we will release the resources in reverse order.

Mechanisms for the lock management (hierarchical lock)

RULES:

- Locks are required starting from the root and down along the tree
- The locks are released starting from the locked node at the smallest granularity and up along the tree
- In order to request a lock IXL, XL or SIXL on a node, we must have a lock IXL or SIXL on the parent node.
- The DBMS respects the lock table.

Mechanisms for the lock management (hierarchical lock)



Request	Resource state				
	ISL	IXL	SL	SIXL	XL
ISL	OK	OK	OK	OK	No
IXL	OK	OK	No	No	No
SL	OK	No	OK	No	No
SIXL	OK	No	No	No	No
XL	No	No	No	No	No

Deadlock (deadlock)

- *deadlock*: when we have concurrent transactions, each of which holds and waits for resources held by others.

Example:

- t_1: *read*(x) *write*(y)
- t_2: *read*(y) *write*(x)

Schedule in 2PL:

$r\_lock_1(x)\ r\_lock_2(y)\ read_1(x)\ read_2(y)\ w\_lock_1(y)\ w\_lock_2(x)$

Deadlock Resolution

- A deadlock corresponds to a cycle in *waiting graph* (Node = transaction, edge = wait for)
- Three techniques
  1. Timeout (Problem: Selection of the range, we need a good balance)
  2. Deadlock Detection
  3. Deadlock Prevention
- Detection: finding cycles in the waiting graph
- Prevention: killing of "suspicious" transactions

Transactions with different levels of isolation

- In SQL-3 the transactions can be defined
- **`read-only`** (they cannot require exclusive locks)
- **`read-write`** (default)

Transactions with different levels of isolation

If we relax the serializability property, we can improve performances, but it is dangerous!

In SQL-92, with SET TRANSACTION it is possible to chose the preferred level of isolation

SET TRANSACTION ISOLATION LEVEL

[READ UNCOMMITTED | READ COMMITTED |

REPEATABLE READ | SERIALIZABLE]

By default, the isolation level is serializable.

Transactions with different levels of isolation

- *Read uncommitted*: also called *dirty read* or *degree of isolation 0*.
  - It does not pose constraints on the locks.
  - It does not require locks for reading and, for reading, does not respect exclusive locks.

  Useful for read-only transactions.

Transactions with different levels of isolation

- *Read committed*: also called *cursor stability* o *degree of isolation 1*.
  - The DBMS accepts requests of shared locks for reading, but without 2PL
  - 2PL used for writing

Transactions with different levels of isolation

- *Repeatable read*: also called *degree of isolation 2*.
  - The DBMS uses the 2PL both for reading and writing, but works only at the tuple level.

It does not prevent the phantom insert.

Transactions with different levels of isolation

- *Serializable*: also called *degree of isolation 3*.
  - Uses the 2PL both for reading and writing at the "predicate" level

Predicate: statement SQL.

Can lead to locks on:

- data
- indexes (in the best case)

Transactions with different levels of isolation

- `read uncommitted:` dirty read, non-repeatable read, phantom read and phantom insert are permitted
- `read committed` dirty read is prevented; non-repeatable read, phantom read and phantom insert are permitted
- `repeatable read` prevents all the anomalies except phantom insert
- `serializable` prevents all the anomalies

- Note
  - Update loss is always prevented

Transactions with different levels of isolation

Low levels require more effort from the developers: ACID properties have to be manually guaranteed.

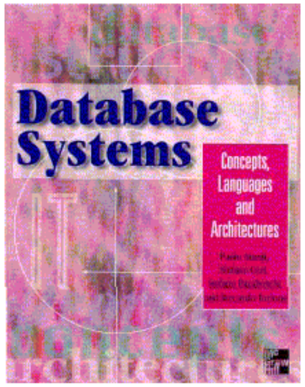
High levels: less efficient, but also less dangerous and less expensive (to implement)

Transactions with different levels of isolation

Some DBMSs (SQL Server, Oracle-not default) also include an additional level: **`snapshot`**.

This is based on the concept of *multiversion* of the data, according to which, during the execution of the transactions which modify the database, two versions are available: before/after the modification. This means that reading operations work on a consistent copy of the database, but not the latest.

References



Database Systems - Concepts, Languages and Architectures
Paolo Atzeni, Stefano Ceri, Stefano Paraboschi and Riccardo Torlone

For more information:

Elmasri, Navathe
Database Systems - Complements, Fourth Edition
Pearson Education, 2005
Chapters: 1.2