Transaction Management

Read Chapter 10 of Riguzzi et al. Sistemi Informativi

Slides derived from those by Atzeni et al.

Transaction

- Elementary unit of work performed by an application
- It is a sequence of SQL statements, usually containing at least one UPDATE, DELETE or INSERT
- An application can contain more than one transaction
- Example: transaction for transferring money from one bank account to another

```
START TRANSACTION;
UPDATE Account
    SET Balance = Balance + 10 WHERE AccountNumber = 12202;
UPDATE Account
    SET Balance = Balance - 10 WHERE AccountNumber = 42177;
COMMIT;
```

Transactions

- In SQL a transaction is started with START TRANSACTION and ended by
  - COMMIT: the transaction has ended successfully
  - ROLLBACK: the transaction has ended unsuccessfully, the work performed must be undone, also called abort

Example

```
START TRANSACTION;
UPDATE Account
    SET Balance = Balance + 10 WHERE AccountNumber= 12202;
UPDATE Account
  SET Balance = Balance – 10 WHERE AccountNumber = 42177;
SELECT Balance INTO A
    FROM Account
    WHERE AccountNumber = 42177;
IF (A>=0)  THEN COMMIT;
               ELSE ROLLBACK;
```

Well-formed Transactions

- One and only one from COMMIT and ROLLBACK must be executed by a transaction
- No operation is performed after COMMIT or ROLLBACK

Transaction

- If no START TRANSACTION is issued, usually every statement is considered a transaction
- Autocommit modality

ACID Properties

- Transactions must possess the ACID properties:
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Atomicity

- A transaction is an atomic unit of execution, it can not be divided
- Either all the operations in the transaction are executed or none is
  - E.g., in the case of the bank transfer, the execution of a single update statement would be disastrous
- The instant in which COMMIT is executed marks the atomic and indivisible instant in which the transaction ends successfully:
  - An error before should cause the rollback of the work performed
  - An error afterwards should not alter the effect of the transaction

Atomicity

- The rollback of the work performed can be caused
  - By a ROLLBACK (“suicide”) statement
  - by the DBMS (“homicide”), for example for the violation of integrity constraints or for concurrency management
- In case of a rollback, the work performed must be undone, bringing the database to the state it had before the start of the transaction

Consistency

- The execution of the transaction must not violate the integrity constraints on the database
- Examples of integrity constraints:
  - A table can not contain two rows with the same value for a column declared as a primary key
  - The sum of the balances of every couple of accounts must remain the same
    - If Balance(a)+Balance(b)=Constant before the transaction then Balance(a)+Balance(b)=Constant after the transaction
    - This type of constraint can be temporarily violated during the execution of the transaction, but must be satisfied at the end

Isolation

- The execution of a transaction must be independent from the concurrent execution of other transactions
- In particular, the concurrent execution of a number of transaction must produce the same result as the execution of the same transactions in a sequence
- E.g. the concurrent exection of T1 and T2 must produce the same results that can be obtained by executing one of these sequences
  - T1,T2
  - T2,T1

Durability

- After a transaction has committed, its modifications to the database must never be lost, they must be persistent
  - Any type of failure after the commit should not alter the modifications

Properties and DBMS Modules

- Atomicity and Durability
  - Reliability manager
- Isolation:
  - Concurrency Manager
- Consistency:
  - Managed by the DDL compilers that introduce the required consistency checks in the code that is executed by the transaction

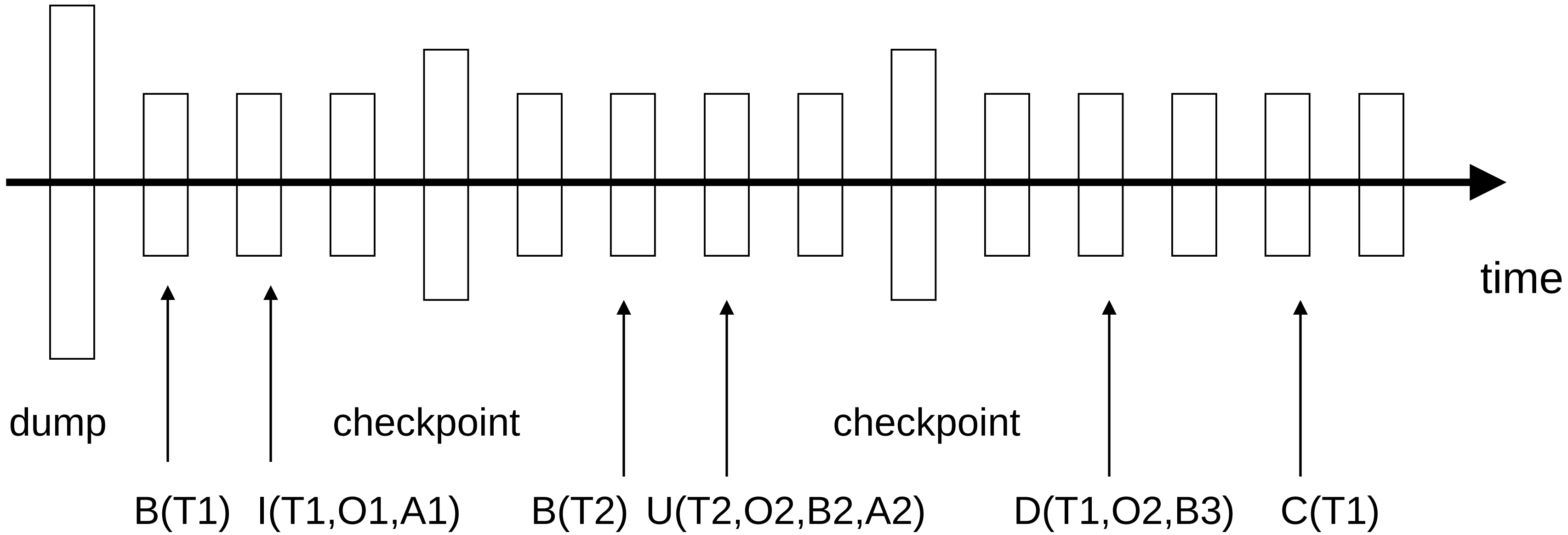
Reliability Manager

- Uses the log, a sequential file where all the operations performed by the transactions are recorded
- This allows the reliability manager to undo or redo the transaction operations
- It must be stored on **stable storage**: it should be more secure than the storage where the data is kept
  - Usually RAID 1 or RAID 5 disks are used
- The reliability manager has the responsibility of executing the START TRANSACTION, COMMIT and ROLLBACK statements

Log

- The reliability manager stores in the log
  - Transaction operations
    - BEGIN, B(T); COMMIT, C(T); ABORT, A(T)
    - INSERT, I(T,O,AS) O=object, AS=After State
    - DELETE, D(T,O,BS) BS=Before State
    - UPDATE, U(T,O,BS,AS)
  - System operations
    - DUMP
    - CHECKPOINT

Exmple of a Log



Undo and Redo

- These records allow the reliability manager to undo or redo operations
- Undo of an operation on an object O
  - UPDATE or DELETE: copy the before state on O
  - INSERT: delete O
- Redo of an operation on an object O
  - UPDATE or INSERT: copy the after state on O
  - DELETE: delete O

Undo and Redo

- They are both idempotent, if A is an operation
  - Undo(Undo(A))=Undo(A)
  - Redo(Redo(A))=Redo(A)

Checkpoint and Dump

- Periodic operations that are needed for recovering from errors
  - Checkpoints for warm and cold restarts
  - Dumps for cold restarts
- Checkpoints are performed by the reliability manager
  - They record all the transactions that are active (i.e. they have not committed nor aborted) at the time of the checkpoint
  - They write the dirty pages of committed transactions in the buffer to disk using a force

Checkpoint

- Simplest version:
  1. No operation is accepted (no updates, deletes, inserts, commits or aborts)
  2. All the dirty pages of committed transaction are transferred to disk with a force
  3. A checkpoint record including the identifiers of the transactions active at the time of the checkpoint is written to the log in a synchronous way (with a force)
  4. New operations are accepted again
- In this way all the committed transactions have their changes stored on disk and the active transactions are stored in the log in the checkpoint record
- We indicate a checkpoint record in the log with CK(T1,T2,…,Tn)

Dump

- A dump is a backup of the whole database
- The backup can be put on
  - A disk different from the one of the database
  - A disk on another machine
  - More commonly, a tape
- After the backup has been performed, a dump record is stored on the log registering the date and time of the backup plus other information

Rules

- The reliability manager enforces two rules:
  - Write Ahead Log (WAL) requires that the log record corresponding to an operation is written to disk before the modified data is written to disk. In this way the modification of an aborted transaction can be undone.
  - Commit Precedence: The log records corresponding to a transactions are written to disk before the commit is executed. In this way the modifications of a committed transaction can be redone.

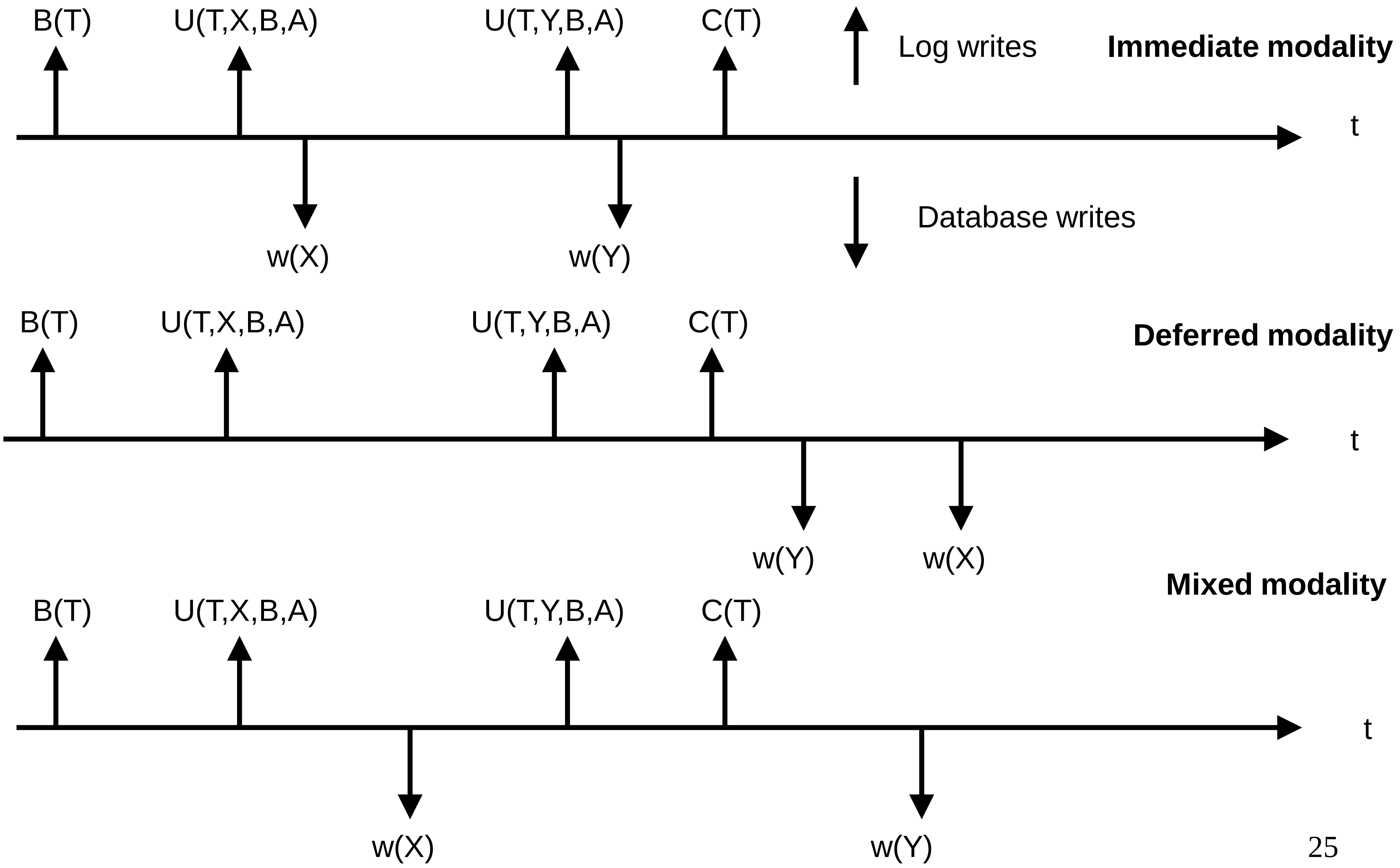
Transactions

- A transaction is considered committed when a commit record is written to disk in the log
  - A failure before this moment results in the Undo of all the operations performed by the transaction
  - A failure after this moment results in the Redo of all the operations performed by the transaction

When Writing Data to Disk?

- Immediate modality: data modifications are written to disk before commit. It doesn’t require Redo but it requires Undo
- Deferred modality: data modifications are written to disk only after commit. It doesn’t require Undo but it requires Redo
- Mixed modality: some data modifications can be written before commit, some after. It requires both Undo and Redo. In this way writes to disk are optimized. This is the scheme most commonly used

When Writing Data to Disk?



Failures

- Two types:
  - “Soft” failures: software bugs, power failures
    - Loss of the primary memory, not of secondary memory (data and log)
    - Warm restart
  - “Hard” failures: data disk failure (e.g. head crash)
    - Loss of the primary memory and the secondary memory where the data is stored but not loss of the secondary memory where the log is stored
    - Cold restart

Failures

- In case of a failure of the disk where the log is stored, we are not able to recover exactly to the point of failure
- A backup must be restored
- The modifications performed after the backup are lost
- Otherwise, no modification performed by a committed transaction is lost

Warm Restart

- Classify transactions at the instant of failure as
  - Completed (committed and with all the data on disk) (those that committed before the last checkpoint)
  - Committed but not completed (Redo is necessary) (those that committed after the last checkpoint)
  - Uncommitted (Undo is necessary) (those that were aborted after the last checkpoint or that did not commit after the last checkpoint and before the failure)

Warm Restart

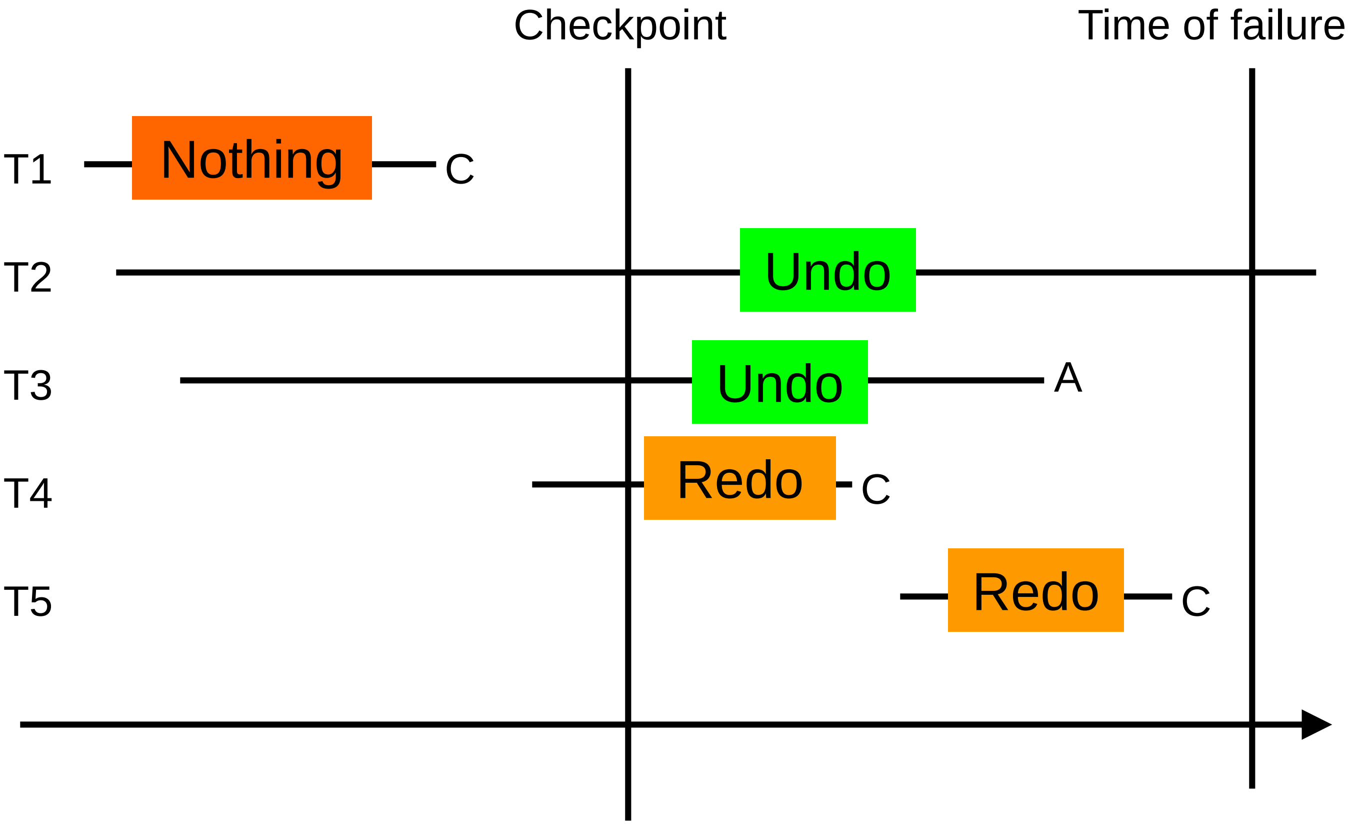
1. Start from the last record of the log and, scanning backwards the log, find the last checkpoint
2. Build the UNDO and REDO sets of transactions identifiers
    1. UNDO is initialized with the transactions active at the time of checkpoint, REDO is initialized with an empty set
    2. Scan the log forward
        - Add to UNDO the transaction with B
        - Move from UNDO to REDO the transaction with C
3. Scan the log backward until the first operation of the oldest transaction in the UNDO and REDO sets, undoing all the operations of the transactions in UNDO (Undo phase)
4. Scan the log forward, redoing all the operations of the transactions in REDO (Redo phase)

Atomicity and Durability

- Atomicity is preserved:
  - transactions active at the time of failure are undone
  - Transactions with an abort record after the last checkpoint are undone
- Durability is preserved:
  - Transactions that committed before the last checkpoint had their pages written to disk at the time of the checkpoint
  - Transactions that committed after the last checkpoint are redone

Example of a Warm Restart

B(T1)
B(T2)
U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
U(T2,O3,B3,A3)
C(T1)
B(T4)
U(T3,O2,B4,A4)
U(T4,O3,B5,A5)
CK(T2,T3,T4)
C(T4)
B(T5)
U(T3,O3,B6,A6)
U(T5,O4,B7,A7)
D(T3,O5,B8)
A(T3)
C(T5)
I(T2,O6,A9)



1. Scan Backward for the Checkpoint

B(T1)
B(T2)
U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
U(T2,O3,B3,A3)
C(T1)
B(T4)
U(T3,O2,B4,A4)
U(T4,O3,B5,A5)
CK(T2,T3,T4)
C(T4)
B(T5)
U(T3,O3,B6,A6)
U(T5,O4,B7,A7)
D(T3,O5,B8)
A(T3)
C(T5)
I(T2,O6,A9)

2. Build the UNDO and REDO Sets

B(T1)
B(T2)
10 U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
9 U(T2,O3,B3,A3)
C(T1)
B(T4)
8 U(T3,O2,B4,A4)
11 U(T4,O3,B5,A5)
1 CK(T2,T3,T4)
2 C(T4)
3 B(T5)
7 U(T3,O3,B6,A6)
12 U(T5,O4,B7,A7)
6 D(T3,O5,B8)
A(T3)
4 C(T5)
5 I(T2,O6,A9)

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

3. Undo Phase

B(T1)
B(T2)
10 U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
9 U(T2,O3,B3,A3)
C(T1)
B(T4)
8 U(T3,O2,B4,A4)
11 U(T4,O3,B5,A5)
1 CK(T2,T3,T4)
2 C(T4)
3 B(T5)
7 U(T3,O3,B6,A6)
12 U(T5,O4,B7,A7)
6 D(T3,O5,B8)
A(T3)
4 C(T5)
5 I(T2,O6,A9)

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

Undo
5 D(O6)
6 I(O5,B8)
7 O3 = B6
8 O2 = B4
9 O3 =B3
10 O1=B1

4. Redo Phase

B(T1)
B(T2)
10 U(T2, O1, B1, A1)
I(T1, O2, A2)
B(T3)
9 U(T2,O3,B3,A3)
C(T1)
B(T4)
8 U(T3,O2,B4,A4)
11 U(T4,O3,B5,A5)
1 CK(T2,T3,T4)
2 C(T4)
3 B(T5)
7 U(T3,O3,B6,A6)
12 U(T5,O4,B7,A7)
6 D(T3,O5,B8)
A(T3)
4 C(T5)
5 I(T2,O6,A9)

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

5 D(O6)
6 I(O5,B8)
7 O3 = B6

Undo

8 O2 = B4
9 O3 =B3
10 O1=B1

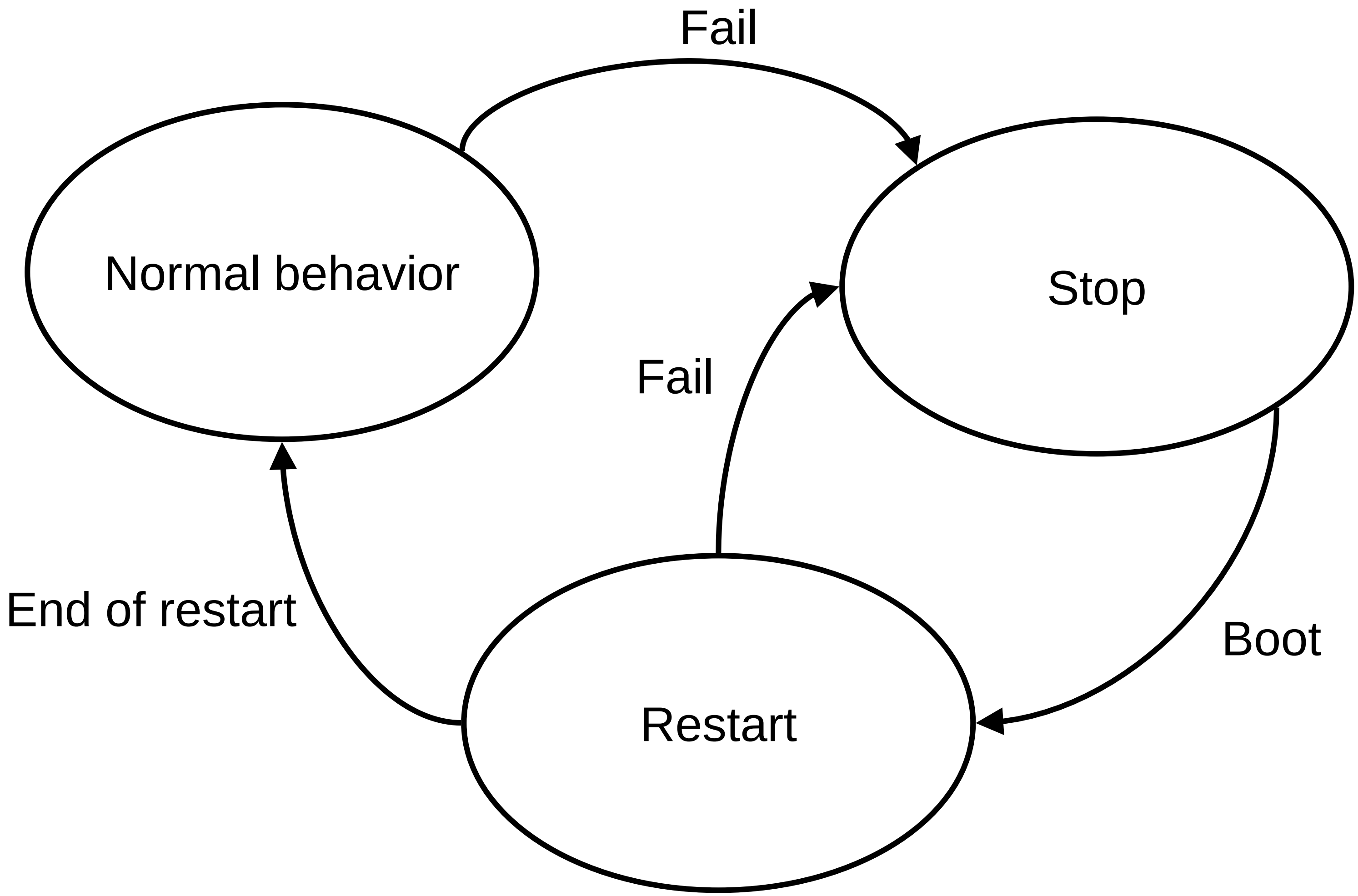
Redo

11 O3 = A5
12 O4 = A7

Cold Restart

1. The last backup of the database is restored, i.e., the data from the backup is copied over the database
2. The last dump record in the log is found
3. The log is scanned forward applying all the operations recorded up to the point of failure
4. A warm restart is performed

Fail-Stop Model



Concurrency Control

- DBMS must support a high number of transaction per second (tps): 10, 100 or 1000 tps
- It is not possible to execute the transactions sequentially
- It is necessary to execute them in parallel
- Abstraction:
  - Database objects are indicated with letters (x, y ,z)
  - Two operations
    - Read x: r(x)
    - Write x: w(x)

Anomalies

- The concurrent execution of transactions can cause a number of anomalies:
  - Lost update
  - Dirty read
  - Inconsistent read (nonrepeatable read)
  - Ghost update
  - Phantom insert

Lost Update

- Two identical transactions (two updates):
  - T1 : r(x), x = x + 1, w(x)
  - T2 : r(x), x = x + 1, w(x)
- Initially x=2, after a sequential execution x=4
- A concurrent execution

T1	T2
bot	
$r_1(x)$	
x = x + 1	
	bot
	$r_2(x)$
	x = x + 1
$w_1(x)$	
commit	
	$w_2(x)$
	commit

- At the end *x*=3, an update is lost

Dirty Read

T1	T2
bot	
$r_1(x)$	
x = x + 1	
$w_1(x)$	
	bot
	$r_2(x)$
abort	
	commit

- Initially x=2, at the end x=2 but T2 has read the value 3 and may have acted on it
- x=3 is an intermediate (“dirty”) value that should not be read by other transactions

Inconsistent Read (Nonrepeatable Read)

T1	T2
bot	
$r_1(x)$	
	bot
	$r_2(x)$
	$x = x + 1$
	$w_2(x)$
	commit
$r_1(x)$	
commit	

- T1 reads twice and reads two different values for x

Ghost Update

- Assume there is a constraint $x + y = 1000$

T1	T2
bot	
$r_1(x)$	
	bot
	$r_2(x)$
	x = x - 100
	$r_2(y)$
	y = y + 100
	$w_2(x)$
	$w_2(y)$
	commit
$r_1(y)$	
s = x + y	
commit	

- At the end s = 1100: the constraint is violated, even if T2 (if fully executed) respects the constraint

Phantom Insert (or Delete)

T1	T2
bot	
“Compute the total number of units of product A sold in the last year”	
	bot
	“insert (or delete) a sale of product A"
	commit
“Compute the total number of units of product A sold in the last year”	
commit	

- The second value computed will be different from the first

Formal Model of a Transaction

- A transaction is modeled as a sequence of read and write operations indexed with the transaction number
- Operations apart from reads and writes do not influence concurrency control
- Example:

$$T1 : r_1(x)\ r_1(z)\ w_1(x)\ w_1(z)$$

- Schedule: a sequence of operations performed by a number of transactions as they are executed by the DBMS
- Example:

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

Scheduler

- The Concurrency Manager is also called Scheduler
- Its job is to accept or reject operation requested by the transactions so that the resulting schedule is acceptable
- **Commit projection** assumption: all the transaction have a commit result. The aborted transactions are not considered in the schedule.
- For the moment we do not consider the dirty read anomaly

Serial and Serializable Schedules

- A schedule is **serial** if the operation of the various transactions are executed in sequence, e.g.

  S_2 : $r_0(x)$ $r_0(y)$ $w_0(x)$ $r_1(y)$ $r_1(x)$ $w_1(y)$ $r_2(x)$ $r_2(y)$ $r_2(z)$ $w_2(z)$
- A schedule is **serializable** if it is equivalent to one of the serial schedules with the same transactions
- It requires a notion of equivalence between schedules
- Two schedules are equivalent if they produce the same result
- A schedule is acceptable if it is serializable
- Impossible to check in practice without executing the schedule. Simpler notions are used.

View Equivalence

- 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 a **final write** if it is the last write of *x* in *S*
- Two schedules S_i and S_j are **view equivalent** ($S_i \approx_V S_j$) if they have the same relation reads from and the same final write for every object x
- A schedule is view serializable if it is view equivalent to a serial schedule
- VSR is the set of view serializable schedules

Examples of View Serializability

- S_3 : $w_0(x)\ r_2(x)\ r_1(x)\ w_2(x)\ w_2(z)$
  - Reads from=$\{(r_2(x),w_0(x)),(r_1(x),w_0(x))\}$
  - Final writes=$\{w_2(x),w_2(z)\}$
- S_4 : $w_0(x)\ r_1(x)\ r_2(x)\ w_2(x)\ w_2(z)$
  - Reads from=$\{(r_1(x),w_0(x)),(r_2(x),w_0(x))\}$
  - Final writes=$\{w_2(x),w_2(z)\}$
- S_3 is view-equivalent to the serial schedule S_4 (so it is view-serializable)

Examples of View Serializability

- S_5 : $w_0(x)$ $r_1(x)$ $w_1(x)$ $r_2(x)$ $w_1(z)$
  - Reads from={($r_1(x)$,$w_0(x)$),($r_2(x)$,$w_1(x)$)}
  - Final writes={$w_1(x)$,$w_1(z)$}
- S_6 : $w_0(x)$ $r_1(x)$ $w_1(x)$ $w_1(z)$ $r_2(x)$
  - Reads from={($r_1(x)$,$w_0(x)$),($r_2(x)$, $w_1(x)$)}
  - Final writes={$w_1(x)$,$w_1(z)$}
- S_5 is view-equivalent to the serial schedule S_6 (so it is view-serializable)

Examples of View Serializability

- S_7 : $r_1(x)$ $r_2(x)$ $w_2(x)$ $w_1(x)$ (lost update)
  - Reads from={}
  - Final writes={$w_1(x)$}
- Serial schedules
  - S_8 : $r_1(x)$ $w_1(x)$ $r_2(x)$ $w_2(x)$
    - Reads from={($r_2(x)$,$w_1(x)$)}
    - Final writes={$w_2(x)$}
  - S_9 : $r_2(x)$ $w_2(x)$ $r_1(x)$ $w_1(x)$
    - Reads from={($r_1(x)$,$w_2(x)$)}
    - Final writes={$w_1(x)$}
- S_7 not view serializable

Examples of View Serializability

- S_{10} : $r^1_1(x)$ $r_2(x)$ $w_2(x)$ $r^2_1(x)$ (inconsistent read)
  - Reads from={($r^2_1(x)$, $w_2(x)$)}
  - Final writes={$w_2(x)$}
- Serial schedules
  - S_{11} : $r^1_1(x)$ $r^2_1(x)$ $r_2(x)$ $w_2(x)$
    - Reads from={}
    - Final writes={$w_2(x)$}
  - S_{12} : $r_2(x)$ $w_2(x)$ $r^1_1(x)$ $r^2_1(x)$
    - Reads from={($r^1_1(x)$,$w_2(x)$),($r^2_1(x)$,$w_2(x)$)}
    - Final writes={$w_2(x)$}
- S_{10} not view serializable

Examples of View Serializability

- S_{13} : $r_1(y)$ $r_2(y)$ $r_2(z)$ $w_2(y)$ $w_2(z)$ $r_1(z)$ (ghost update)
  - Reads from={($r_1(z)$,$w_2(z)$)}
  - Final writes={$w_2(y)$,$w_2(z)$}
- Serial schedules
  - S_{14} : $r_1(y)$ $r_1(z)$ $r_2(y)$ $r_2(z)$ $w_2(y)$ $w_2(z)$
    - Reads from={}
    - Final writes={$w_2(y)$,$w_2(z)$}
  - S_{15} : $r_2(y)$ $r_2(z)$ $w_2(y)$ $w_2(z)$ $r_1(y)$ $r_1(z)$
    - Reads from={($r_1(y)$,$w_2(y)$),($r_1(z)$,$w_2(z)$)}
    - Final writes={$w_2(y)$,$w_2(z)$}
- S_{13} not view serializable

View Serializability

- To determine whether two schedules are view equivalent has linear complexity
- To determine whether a schedule is view serializable is an NP-hard problem
  - It can not be used in practice

Conflict Equivalence

- Definition:
  - Given two operations a_i *and* a_j ($i \neq j$), a_i is in *conflict* with a_j ($i \neq j$), if they work on the same object and at least one of them is a write. Two cases:
    - Conflict *read-write* (*rw* o *wr*)
    - Conflict *write-write* (*ww*).
- Two schedules S_i and S_j are **conflict equivalent** ($S_i \approx_C S_j$) if every couple of actions that are in conflict appear in the same order in the two schedules, i.e., if the conflicts relation is the same
- A schedule is conflict serializable if it is conflict equivalent to a serial schedule
- CSR is the set of conflict serializable schedules

Example of Conflict Serializability

- S_{16}= $w_0(x)$ $r_1(x)$ $w_0(z)$ $r_1(z)$ $r_2(x)$ $r_3(z)$ $w_3(z)$ $w_1(x)$
- Conflicts={$(w_0(x),r_1(x)),(w_0(x),r_2(x)),(w_0(x),w_1(x)),$
  $(w_0(z),r_1(z)),(w_0(z),r_3(z)),(w_0(z),w_3(z)),(r_1(z),w_3(z)),$
  $(r_2(x),w_1(x))$ }
- S_{17}=$w_0(x)$ $w_0(z)$ $r_2(x)$ $r_1(x)$ $r_1(z)$ $w_1(x)$ $r_3(z)$ $w_3(z)$
- Conflicts={$(w_0(x),r_2(x)),(w_0(x),r_1(x)),(w_0(x),w_1(x)),$
  $(w_0(z),r_1(z)),(w_0(z),r_3(z)),(w_0(z),w_3(z))$, $(r_2(x),w_1(x)),$
  $(r_1(z),w_3(z))$}
- S_{16} is conflict-equivalent to the serial schedule S_{17}

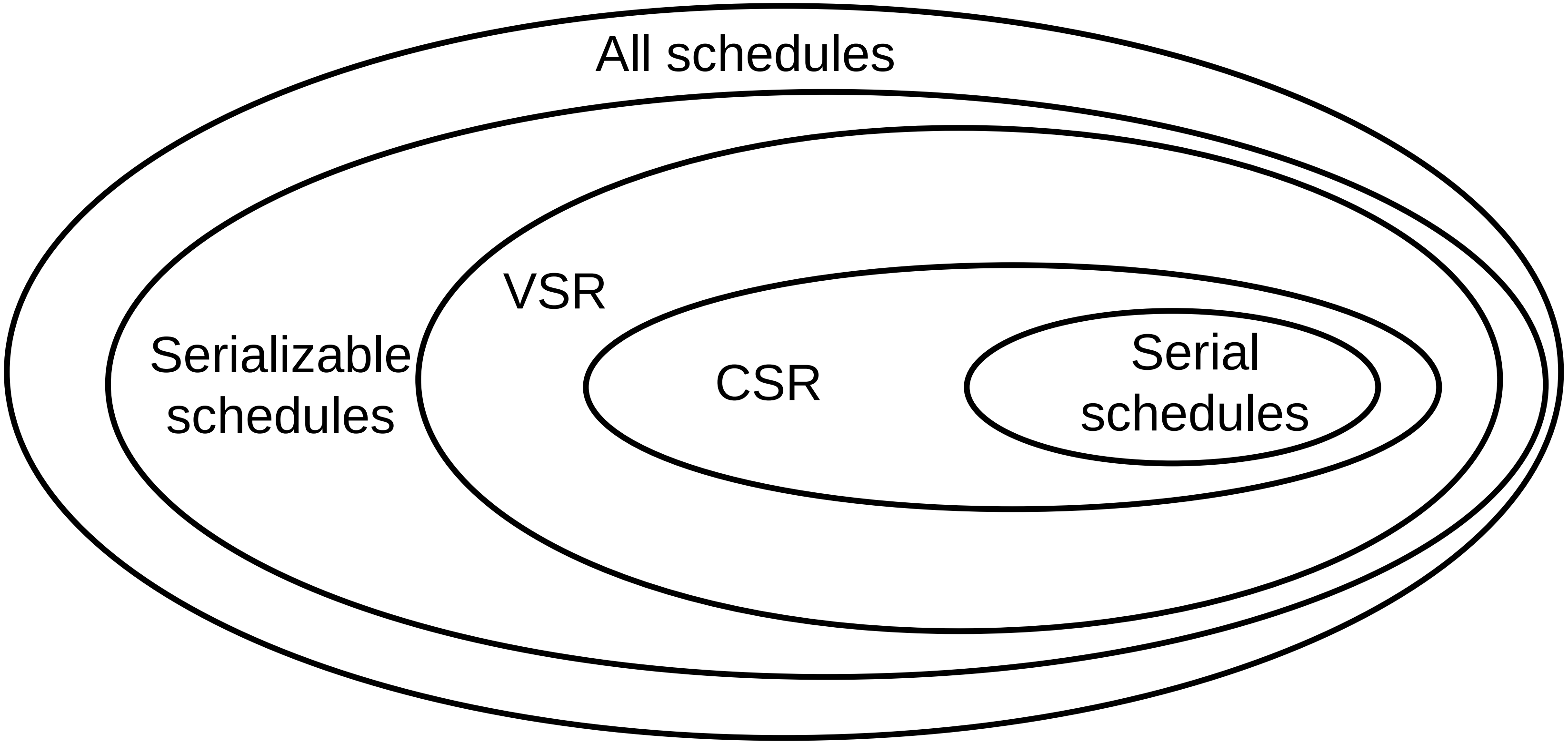
VSR and CSR

- Every schedule conflict-serializable is view-serializable, but not viceversa
  - CSR $\subset$ VSR
- Example of CSR$\neq$VSR:
- $r_1(x)\ w_2(x)\ w_1(x)\ w_3(x)$
  - view-serializable: view-equivalent to $r_1(x)\ w_1(x)\ w_2(x)\ w_3(x)$
  - not conflict-serializable

Proof that CSR $\subseteq$ VSR

- $S_1 \in CSR \Rightarrow \exists\, S_2$ serial such that $S_1 \approx_C S_2$
- We prove that conflict-equivalence $\approx_C$ implies view-equivalence $\approx_V$, i.e. $S_1 \approx_C S_2 \Rightarrow S_1 \approx_V S_2$
- Suppose $S_1 \approx_C S_2$
  S_1 and S_2 have:
  - The same final writes: otherwise, there would be two writes in a different order and, since two writes are in conflict, S_1 and S_2 would not be $\approx_C$
  - The same relation reads from: otherwise, there would be two writes in a different order or a write and a read in a different order and S_1 and S_2 would not be $\approx_C$

CSR and VSR


All schedules
Serializable schedules
VSR
CSR
Serial schedules

Test of Conflict Serializability

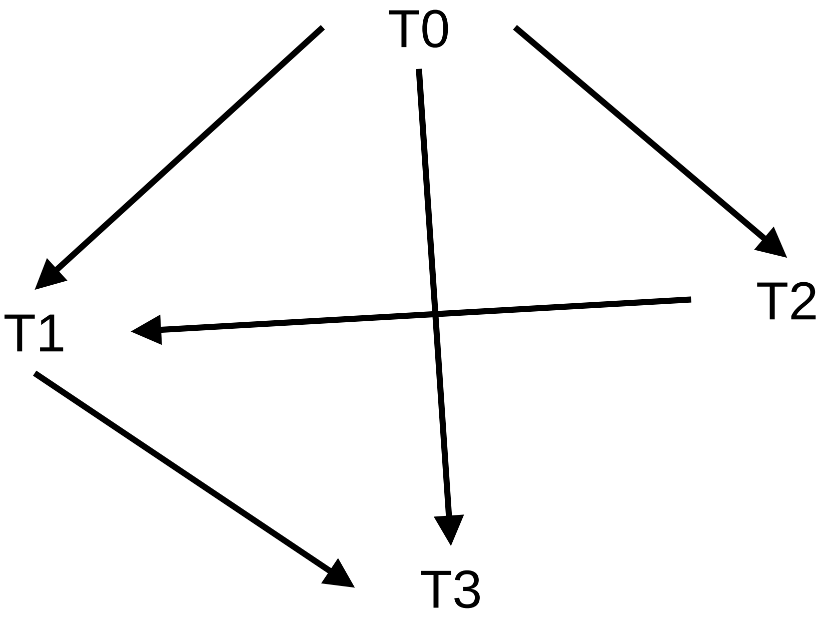
- By means of the conflict graph G:
  - A node for every transaction
  - An arc from transaction Ti to transaction Tj if there exist at least one conflict between an action a_i of Ti and an action a_j of Tj and a_i precedes a_j
- $S \in CSR \Leftrightarrow$ G is acyclic

Proof of $S \in CSR \Leftrightarrow$ G is acyclic

- If $S \in CSR$ then $S \approx_C$ to a serial schedule S_s. Suppose that the transaction in S_s are ordered according to the TID: T1, T2, …, Tn. Since S_s has all the actions involved in conflicts in the same order as in S, in G there can be only arcs (i,j) with i<j so G can't have cycles because a cycle requires at least one arc (i,j) with i>j
- If G is acyclic, then it exists a topological order among the nodes, i.e. an assignment of integers to nodes such that G contains only arcs (i,j) with i<j. The serial schedule whose transactions are ordered according to the topological order is $\approx_C$ to S because for every conflict (i,j) we have i<j

Example of a conflict graph

S_{16}= $w_0(x)$ $r_1(x)$ $w_0(z)$ $r_1(z)$ $r_2(x)$ $r_3(z)$ $w_3(z)$ $w_1(x)$



- The graph is acyclic, S_{16} is conflict serializable

Practical Considerations

- Checking the acyclicity of a graph has a linear complexity, so the test of conflict serializability is linear.
- However, it is still not practically viable because the graph must be built at runtime and continually modified
  - Every time an action is requested to the scheduler, it must update the graph and test for acyclicity
- Moreover, we are still under the commit projection assumption
- In practice locking is used

Lock

- New operations:
  - r_lock(x): sets a read lock on x (shared lock)
  - w_lock(x): sets a write lock on x (exclusive lock)
  - unlock(x): frees any locks on x
- A read action r(x) to be successfull must be preceded by a r_lock(x) or a w_lock(x) and followed by an unlock(x)
- A write action w(x) to be successfull must be preceded by a w_lock(x) and followed by an unlock(x)
- An object can be read locked by multiple transactions (shared lock) but can be write locked only by a single transaction (exclusive lock)

Lock operations

Successful?/ New Resource state	Previous resource state		
Request	free	r_locked	w_locked
r_lock	YES / r_locked	YES / r_locked	NO / w_locked
w_lock	YES / w_locked	NO / r_locked	NO / w_locked
unlock	error	YES / depends	YES / free

- When a lock request is not successfull, the transaction is put in a queue and has to wait
- The waiting ends when the resource becomes free and there are no other transactions in the front of the queue
- A counter is kept for each resource that contains the number of transactions having a read lock on the resource
- An unlock operation on a r_locked resource decrements the counter and frees the resource only if the counter goes to 0

Locking

- Information about locks are stored in a lock table
- The scheduler in this case is also called a lock manager
- When a transaction has to read and write a resource, can request a read lock and then increment it to a write lock or request a write lock from the beginning
- Locking prevents conflicts on resource access but doesn’t ensure serializability
- A further condition is required: Two Phase Locking

Two Phase Locking (2PL)

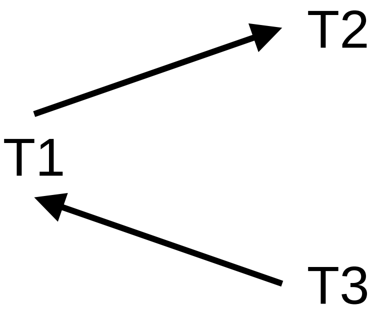
- **A transaction, after having released a lock, can not acquire other locks**
- Thus, every transaction goes through two phases:
  - Lock acquisition (growing phase)
  - Lock release (shrinking phase)
- 2PL is the set of schedules respecting two phase locking
- It is used in practice

$2PL \subset CSR$

- Every schedule that respects two phase locking is also conflict serializable, but not vicecersa
- Example of $2PL \neq CSR$:

$$r_1(x)\ w_1(x)\ r_2(x)\ w_2(x)\ r_3(y)\ w_1(y)$$

  - Not in 2PL
  - Conflict-serializabile



Proof that 2PL $\subseteq$ CSR

- S $\in$ 2PL
- Suppose that S $\notin$CSR
- Then there is a cycle in G: T1, T2, …Tn, T1
- Since there is a conflict between T1 and T2, there is a resource x on which the transactions conflict. For T2 to proceed it is necessary that T1 releases its lock on x
- Since there is a conflict between Tn and T1, there is a resource y on which the transactions conflict. For T1 to proceed it is necessary for T1 to acquire a lock on y
- Thus S does not respect two phase locking, against the hypothesis.
- Thus S $\in$ CSR

Anomalies

- Two phase locking avoids all the anomalies apart from dirty read and phantom insert

Example: Ghost Update

T1	T2	x	y
bot		free	free
	bot		
	$w\_lock_2(x)$	2:write	
$r\_lock_1(x)$		1:wait	
	$r_2(x)$		
	$x=x-100$		
	$w\_lock_2(y)$		2:write
	$r_2(y)$		
	$y=y+100$		
	$w_2(x)$		
	$w_2(y)$		
	commit		
	$unlock_2(x)$	1:read	
$r_1(x)$			
	$unlock_2(y)$		free
$r\_lock_1(y)$			1:read
	eot		
$r_1(y)$			
$s=x+y$			
commit			
eot			

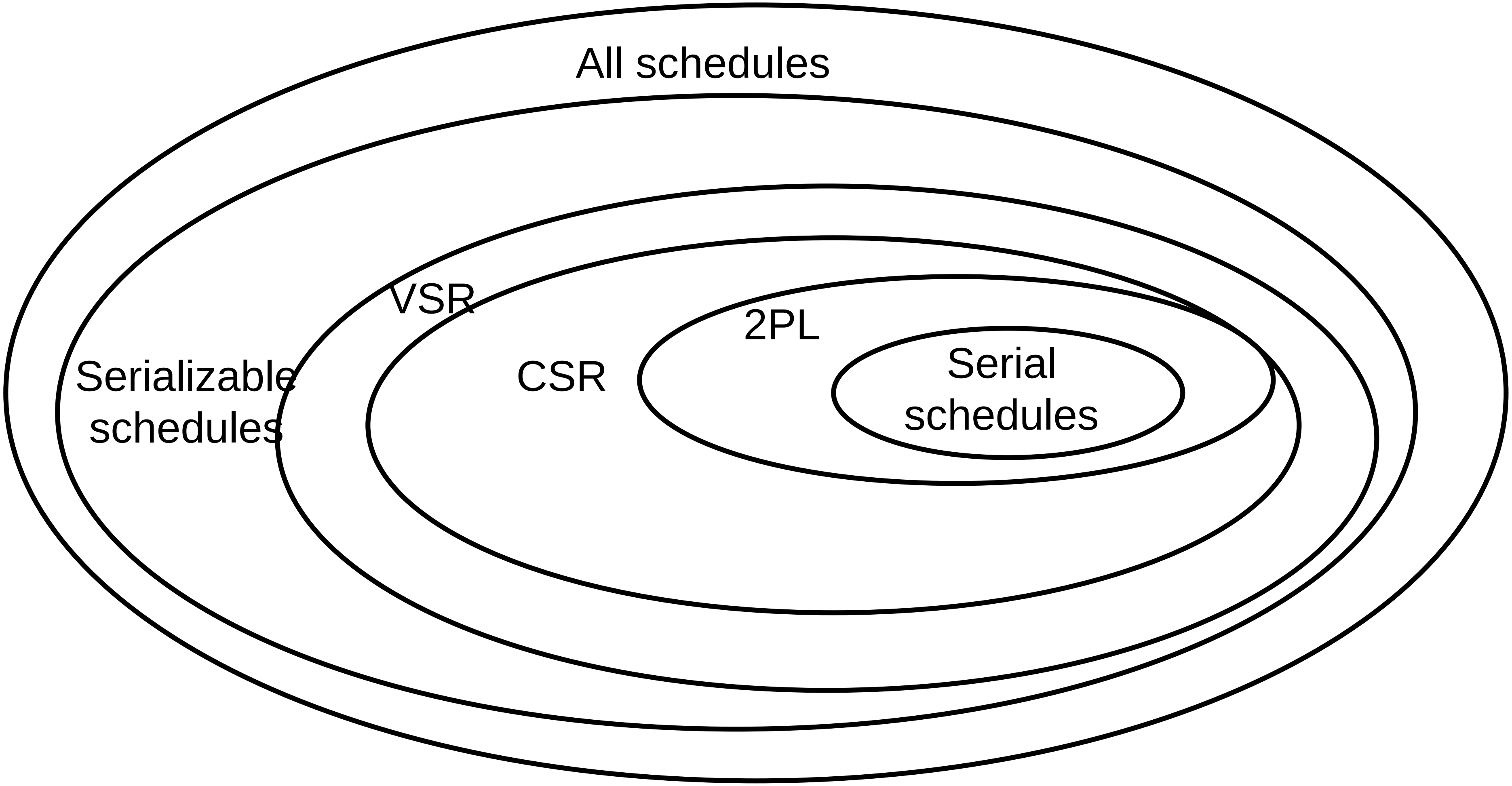
Dirty Read

- The dirty read anomalies remain, because we have used up to now the commit projection assumption
- In order to remove that assumption and to avoid dirty reads, we use a restriction of two phase locking:
- **Strict Two Phase Locking (STPL): a transaction can release the locks only after commit or abort**
- This is the concurrency control method used in practice
- With STPL the dirty read anomaly can not occur because data written by a transaction can be read by another only after the transaction has committed

Phantom Insert (or Delete)

- To avoid phantom insert we need a different type of locks:
  - The locks desribed up to now consider only objects already present in the database
  - To avoid phantom insert we need a lock that depends on a condition (**predicate lock**):
    - E.g. We want to lock all the tuples of the table Sales relative to product A
- Predicate locks are implemented by using indexes or, in case there are no indexes, the whole table is locked

Sets

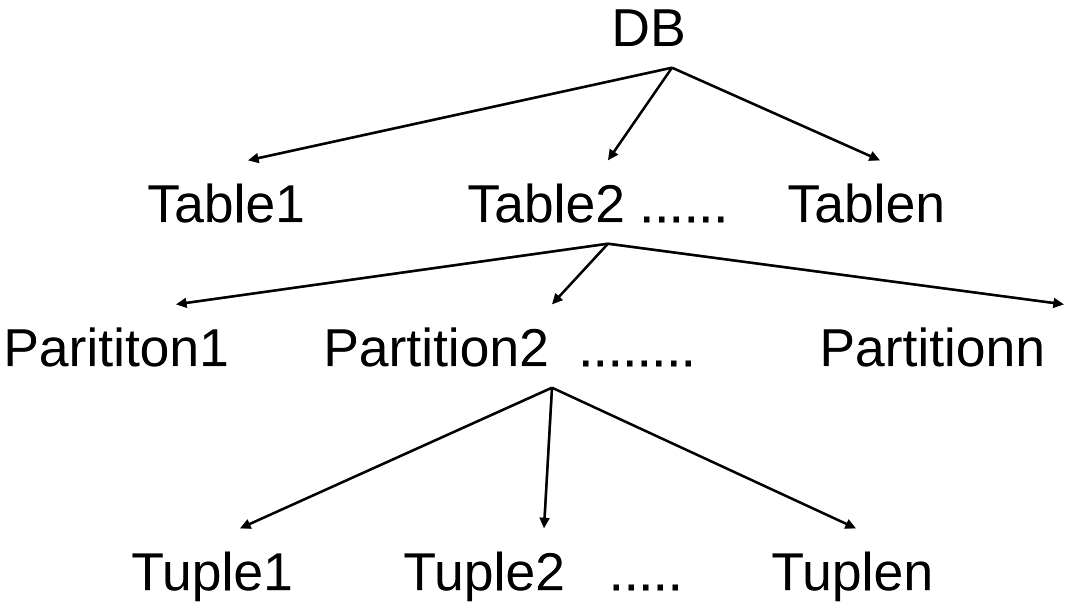

All schedules
Serializable schedules
VSR
CSR
2PL
Serial schedules

Lock Management

- The lock manager offers the following functions
  - r_lock(T,x,errcode,timeout)
  - w_lock(T,x,errcode,timeout)
  - unlock(T,x)
- If a request can not be satisfied, the transaction is put in a queue associated with the resource and the process has to wait
- When a resource is released, the lock manager assigns the resource to the first transaction in the queue
- If a timeout expires, the transaction can request a rollback or request again the lock
- Lock table: for each resource, two state bits and a counter

Hierarchical Locks

- Different granularity levels for the locks



DB

Table1 Table2 ...... Tablen

Parititon1 Partition2 .... Partitionn

Tuple1 Tuple2 ..... Tuplen

Hierarchical Locks

- More locks:
  - XL: exclusive lock (=write lock)
  - SL: shared lock (=read lock)
  - ISL: intentional shared lock, intention of shared locking a descendent of the current node
  - IXL: intentional exclusive lock, intention of exclusive locking a descendent of the current node
  - SIXL: shared intentional-exclusive lock. Current node locked in an shared way, intention of exclusive locking a descendant

Hierarchical Locks

- To ask for a SL or ISL lock on a node you need to have a SL or ISL lock on the parent.
- To ask for an IXL, XL or SIXL lock on a node you need to have a SIXL or IXL lock on the parent
- Example: to exclusively lock a tuple you need:
  - Lock the root with IXL
  - Lock the table with IXL
  - Lock the partition with IXL
  - Lock the tuple with XL
- At the end, the locks must be released in the opposite order

Hierarchical Locks

- Compatibility rules:

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

Hierarchical Locks

- Choice of the lock level
- Tradeoff:
  - Too high level locks can reduce parallelism
  - Too low level locks cause a high overhead of the lock manager

Deadlock

- Caused by the waiting requirements of 2PL
- Example: two transactions lock each a resource and wait for the other to unlock its resource
  - T1: r(x), w(y)
  - T2: r(y), w(x)
  - Schedule:

    $r\_lock_1(x)$, $r\_lock_2(y)$, $r_1(x)$, $r_2(y)$ $w\_lock_1(y)$, $w\_lock_2(x)$

Deadlock Avoidance

- Three techniques:
  1. Timeout
  2. Deadlock prevention
  3. Deadlock detection

Timeout

- Wait only for a fixed amount of time, if after it the resource is not free then the transaction is aborted.
- Tradeoff in the choice of timeout:
  - too low may cause abort also when a deadlock did not occur
  - too high may waste time

Deadlock prevention

- Request of all the resources at the same time
- Timestamp: the wait of Ti on Tj is allowed only if i<j. Otherwise a transaction is killed. Strategy types
  - Preemptive strategy: may kill the transaction that owns the resource
  - Nonpreemptive: kills only the transaction making the request.

Deadlock prevention

- Possible strategy: kill the transaction that has done the least work (preemptive strategy)
- Problem: a transaction using at the beginning a higly contended resource may be repeatedly killed (starvation) since it is always the transaction that has done the least work
- Solution: use the same timestamp when a transaction is restarted. In this way it will eventually wait for the resource. If it gets the resource and a younger transaction requests it that has made the same amount of work, higher priority is assigned to older transaction and the younger transaction is killed
- Infrequent in DBMS due to the high number of killed transaction, deadlock detection better because of the rarity of deadlock

Deadlock detection

- Wait graph: nodes->transactions, arcs->waits
- Search for a cycle in the wait graph, at fixed intervals or when a wait timeout expires
- If a cycle is identified, a transaction is killed

Concurrency Management in SQL:1999

- Transactions can be defined read only (only shared locks) or read write
- The isolation level can be chosen for each transaction:

	Lost updates	Dirty reads	Inconsistent reads	Ghost updates	Phantom inserts
`read uncommitted`	no	yes	yes	yes	yes
`read committed`	no	no	yes	yes	yes
`repeatable read`	no	no	no	no	yes
`serializable`	no	no	no	no	no

Isolation Levels

- On writes we always have Strict 2PL (thus Lost updates are always avoided)
- **`read uncommitted`**:
  - Does not require shared locks for reads and does not respects exclusive locks from other transactions for reads (useful for read only transactions)
- **`read committed`**:
  - Requires shared locks for reads (and respect others'), but without 2PL
- **`repeatable read`**:
  - Strict 2PL also for reads, with locks on tuples
- **`serializable`**:
  - Strict 2PL with predicate locks