Transaction management

Tecnologie delle Basi di Dati M

What’s a transaction?

- A transaction is a logical processing corresponding to a series of elementary physical operations (reads/writes) on the DB
- Examples:
  - Transfer of a sum between bank accounts

```
UPDATE CC                    UPDATE CC
SET balance=balance-50       SET balance=balance+50
WHERE account=123            WHERE account=235
```

  - Updating wages of employees in a branch

```
UPDATE Emp
SET wage=1.1*wage
WHERE  branch=‘S01’
```

Properties

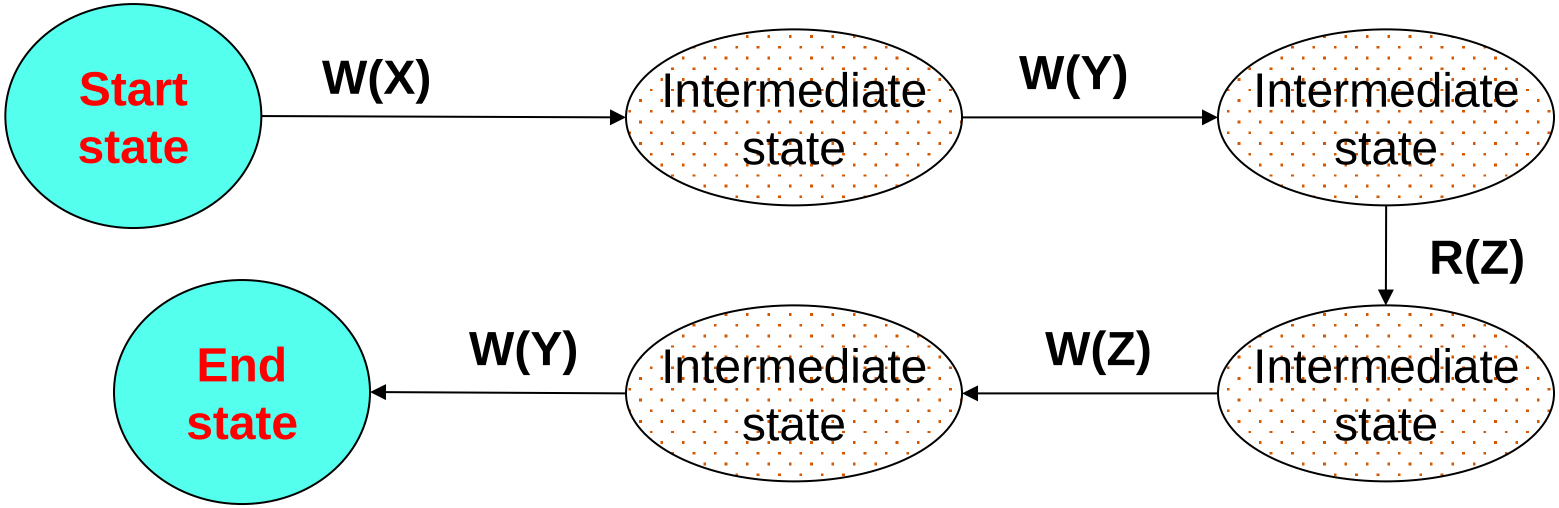
- The ACID acronym denotes the 4 properties that the DBMS should guarantee for every transaction:
  - Atomicity: a transaction is a processing unit
    - The DBMS guarantees that the transaction is performed as a whole
  - Consistency: a transaction leaves the DB in a consistent state
    - The DBMS guarantees that no integrity constraint is violated
  - Isolation: a transaction is executed independently of the others
    - If more than transaction is executed concurrently, the DBMS guarantees that the net effect is equivalent to one of the many possible sequential executions of the same transactions
  - Durability: effects of a correctly terminated transaction should persist over time
    - The DBMS protects the DB against failures

ACID properties and DBMS modules

- Transaction Manager
  - Coordinates the execution of transactions, receiving relevant SQL commands
- Logging & Recovery Manager
  - Is in charge of Atomicity and Durability
- Concurrency Manager
  - Guarantees Isolation
- DDL Compiler
  - Generates code for controlling Consistency

Transaction model

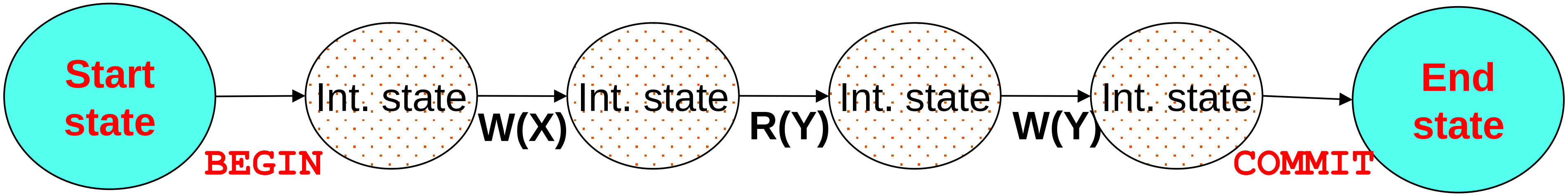
- In the model we consider a transaction is viewed as a sequence of elementary read (R) and write (W) operations on objects (tuples) of the DB that, starting from an initial DB state, brings the DB to a new consistent state



- In general, it is not required that intermediate DB states are consistent

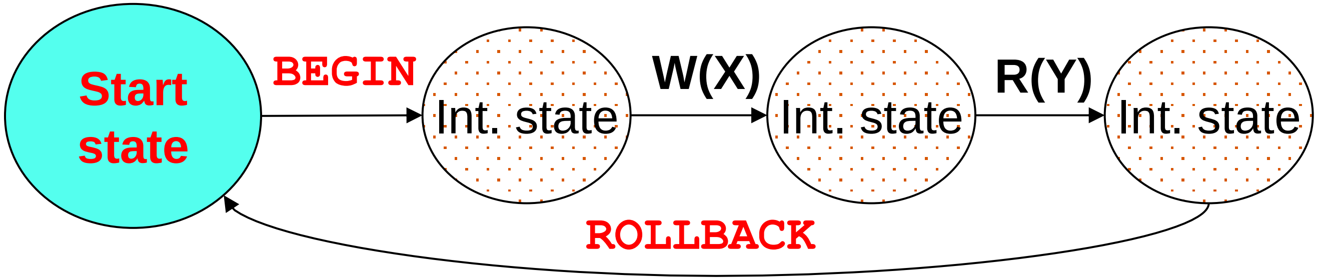
Possible outcomes for a transaction (1)

- In the model we consider, a transaction (whose beginning is specified by the keyword BEGIN, although this is implicit in SQL) can only have two outcomes:
  - Complete successfully:
    This happens only when the transaction, after having executed all its operations, specifies a particular SQL statement, called COMMIT (or COMMIT WORK), that "formally" communicates the successful completion to Transaction Manager



Possible outcomes for a transaction (2)

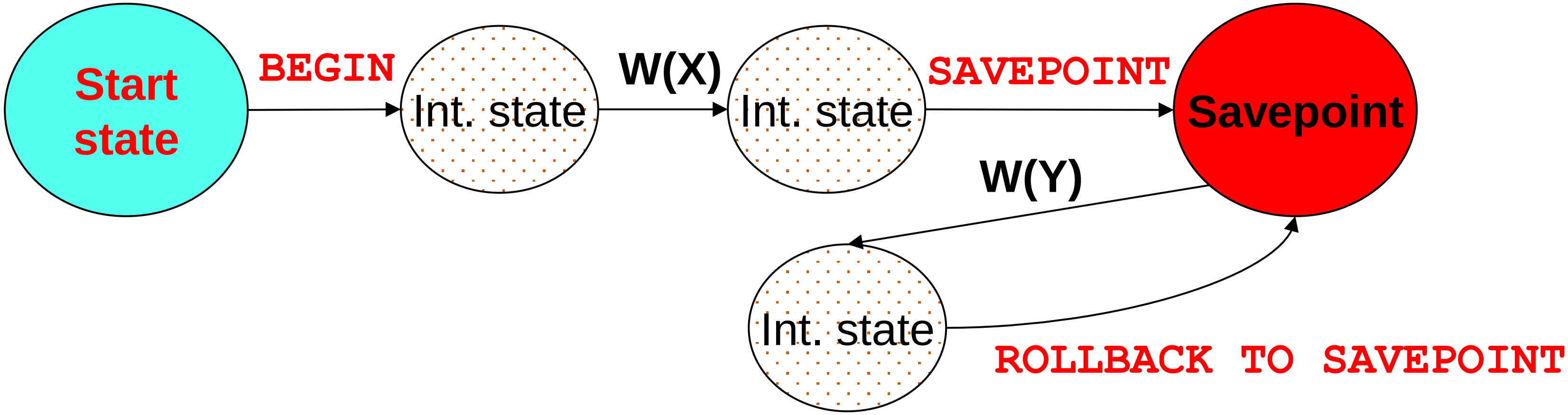
- Complete unsuccessfully (beforehand); 2 cases are possible:
  - The transaction itself, for some reason, decides that it makes no sense to continue and thus “aborts” executing the SQL statement ROLLBACK (or ROLLBACK WORK)
  - The system (e.g., due to a failure or to a constraint violation)cannot guarantee the successful execution of the transaction, which is thus aborted



  - If, for some reason, the transaction is unable to complete successfully, the DBMS should “undo” any change possibly made to the DB

Transactions with savepoint

- The transaction model used by DBMS is actually more complex; in particular, it is possible to define some “savepoint”, which can be used to undo the operations of a transaction only partially



- To define a savepoint in DB2 we use the command
  `SAVEPOINT <name> ON ROLLBACK RETAIN CURSORS`
  to execute a partial rollback
  `ROLLBACK WORK TO SAVEPOINT <name>`

Serial execution of transactions

- Since a DBMS should be able to execute different transaction accessing to shared data, it could execute such transactions in sequence (serial execution)
- E.g., two transactions T1 and T2 could be executed as follows, where the temporal succession of elementary operations on the DB (schedule) is highlighted:

T1	R(X)	W(X)	Commit			
T2				R(Y)	W(Y)	Commit

Concurrent execution of transactions

- Alternatively, the DBMS could execute multiple transactions concurrently, interleaving operations of one transaction with those of other transactions (interleaved execution)
- Concurrent execution of multiple transactions is the key to guarantee performance :
  - We can exploit the fact that, when a transaction is waiting for an I/O operation to complete, another transaction can use the CPU, thus increasing the system “throughput” (no. of transactions processed in the time unit)
  - If we have one “short” and one “long” transactions, interleaved execution reduces the average response time of the system

T1	R(X)				W(X)	Commit
T2		R(Y)	W(Y)	Commit		

Reducing the response time

- T1 is "long", T2 is "short" (for simplicity, every table row represents a time unit
  - Let us suppose that T2 begins at time=2

time	1	2	…	999	1000	1001	1002	1003	1004
T1	R(X1)	W(X1)		R(X500)	W(X500)	Commit			
T2							R(Y)	W(Y)	Commit

- Average response time = (1001 + (1004-1))/2 = 1002

time	1	2	3	4	5	…	1002	1003	1004
T1	R(X1)	W(X1)					R(X500)	W(X500)	Commit
T2			R(Y)	W(Y)	Commit				

- Average response time = (1004 + 3)/2 = 503.5

Isolation: managing concurrency

- The Transaction Manager should guarantee that concurrently executing transactions do not interfere with each other
- If this is not the case, 4 basic types of problems could arise:
  - Lost Update: concurrent updates
  - Dirty Read: reading uncommitted data
  - Unrepeatable Read: interleaving reads and writes
  - Phantom Row: new data not appearing in the result of a query

Examples of isolation problems

- Lost Update: two people, in different shops, buy the very last ticket for the U2 concert in Rome (!?)
- Dirty Read: the U2 tour schedule shows a date in Bologna on 15/07/17, but when you try to buy the ticket for that concert the system tells that no such date exists (!?)
- Unrepeatable Read: for the U2 concert (finally, the date has been decided!) you see a price of 90 €, you think about it a little, but when you're decided, the price is risen to 110 € (!?)
- Phantom Row: you want to go see both U2 concerts in Italy, but when you try to buy tickets, you discover that there are now three dates (!?)

Lost update

- The following schedules show a typical lost update case, where we also highlight operations updating the value of X and show how the value of X in the DB varies over time

This update is lost

T1	R(X)	X=X-1			W(X)	Commit		
X	***1***	1	1	1	***0***	0	***0***	0
T2			R(X)	X=X-1			W(X)	Commit

  - The problem arises because T2 reads the value of X before T1 (that already read it) updates it ("both transactions see the last ticket")

Dirty read

- In this case, the problem arises because a transactions read a value that is not correct:

T1	R(X)	X=X+1	W(X)		Rollback			
X	***0***	0	***1***	1	***0***	0	***0***
T2				R(X)		…	Commit

This read
is "dirty"

  - What T2 does is based on an "intermediate", non-stable value of X, ("the definitive date is not 15/07/17")
  - Consequences are unpredictable(it depends on what T2 does) and would be present even if T1 would not abort

Unrepeatable read

- Now the problem is that a transaction reads a value twice, with different outcomes (“meanwhile, the price has increased”):

T1		R(X)	X=X+1	W(X)	Commit		
X	*0*	0	0	1	*1*	1	1
T2	R(X)					R(X)	Commit

The two reads are inconsistent

- Also in this case serious consequences could arise
- The same problem can occur for “analysis” transactions
  - For example, T1 sums the balance of 2 accounts while T2 transfers money between the two (T1 could report an incorrect total value)

Phantom row

- This case could arise only when tuples are deleted or inserted that should be logically considered by another transaction
  - E.g.: record r4 is “phantom”, since T1 “dose not see it”

T1	R(r2)	R(r3)	…	W(r2)	W(r3)		Commit	
T2				R(X)		Insert(r4)		Commit

T1 cannot see this record

- T1:
  ```
  UPDATE Proj
  SET location=‘Firenze’
  WHERE location=‘Bologna’
  ```
- T2:
  ```
  INSERT INTO Proj
  VALUES(‘P03’,‘Bologna’)
  ```

Properties of a schedule

- Serial: a schedule with transactions executed sequentially
- Serializable: a schedule involving only committed transactions whose effect on any consistent DB instance is guaranteed to be identical to that of some serial schedule
- Recoverable: a schedule where, if transaction T1 reads a change made by transaction T2, T1 commits only after T2 commits
- Cascadeless: a where every transaction can only read changes of committed transactions
- Strict: a schedule where every transaction does not read or write values changed by any other active transaction

Guaranteeing isolation

- A technique commonly used by DBMSs to avoid previous problems consists in locks
  - Locks are a mechanism normally used by operating systems to regulate access to shared resources
  - Before executing any operation, it is required to “acquire” a lock on the requested resource (e.g., a record)
  - The lock request is implicit, thus invisible at SQL level
    - … but we will see that we can do something with SQL, anyway

Lock types

- Locks come in different “flavors” (DB2 has 11 types!)
- The basic ones are:
  - S (Shared): a shared lock is required for reading a value
  - X (eXclusive): an exclusive lock is required to write/update a value

Lock compatibility

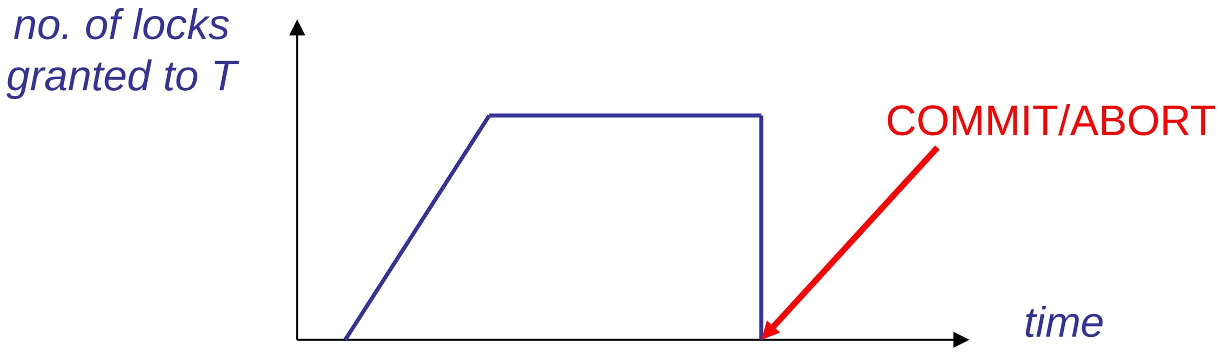
- The Lock Manager is a DBMS module in charge of keeping track which resources are currently used and which transactions are using them (and how)
- When a transaction T wants to operate on a value Y, a lock request on Y is sent to the Lock Manager
- Lock is granted to T according to the following compatibility table

		Another transaction has on Y a lock of type	
		S	X
T requests a lock of type	S	OK	NO
	X	NO	NO

- When T finishes using Y, can release the lock (unlock(Y))

Strict 2-phase lock (Strict 2PL) protocol

- The way transaction release acquired locks is the key to solve concurrency problems
- It can be proven that isolation is guaranteed if:
  - A transaction first acquires all necessary locks
  - Locks are released only at the end of the execution (COMMIT or ABORT)



- As a collateral effect, deadlocks (stalemate situation) can happen, which are solved by aborting a transaction

Preventing lost update

- Previous schedule is modified as follows:

T1	S-lock(X)	R(X)	X=X-1				X-lock(X)	wait	wait
X	*1*	1	1	1	1	1	1	1	1
T2				S-lock(X)	R(X)	X=X-1		X-lock(X)	wait

- Neither T1 nor T2 succeed in acquiring the lock needed to update X (they remain in “wait” state)
- We thus have a deadlock
  - If the DBMS chooses to abort, say, T2, then T1 can proceed

Preventing dirty read

- In this case, correct execution requires that T2 awaits T1 termination before reading the value of X

T1	S-lock(X)	R(X)	X=X+1	X-lock(X)	W(X)		rollback	unlock(X)	
X	***0***	0	0	0	***1***	1	***0***	0	0
T2						S-lock(X)	wait	wait	R(X)

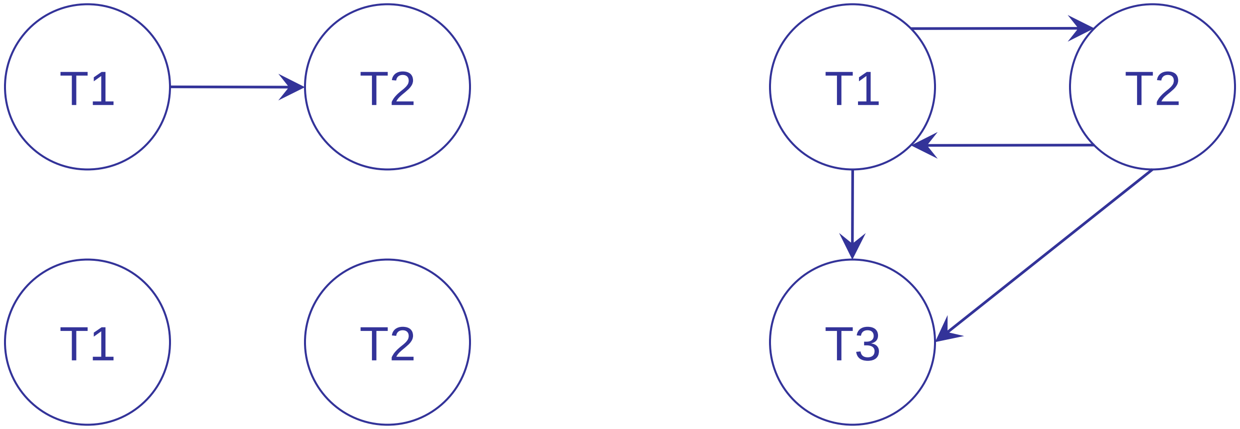
Preventing unrepeatable read

- Also in this case, T2 is put on hold, and T1 is therefore guaranteed to read always the correct value of X

T1			S-lock(X)	R(X)	X=X+1	X-lock(X)	wait	wait	wait	W(X)
X	***0***	0	0	0	0	0	0	0	0	***1***
T2	S-lock(X)	R(X)					R(X)	commit	unlock(X)	

Serializability graph

- Captures all potential conflicts between transactions in a schedule
  - A node for each committed transaction
  - An arc between transactions T_i and T_j if an action of T_i precedes and conflicts with an action of T_j (that is, both act on the same data object and at least one of them is a write)

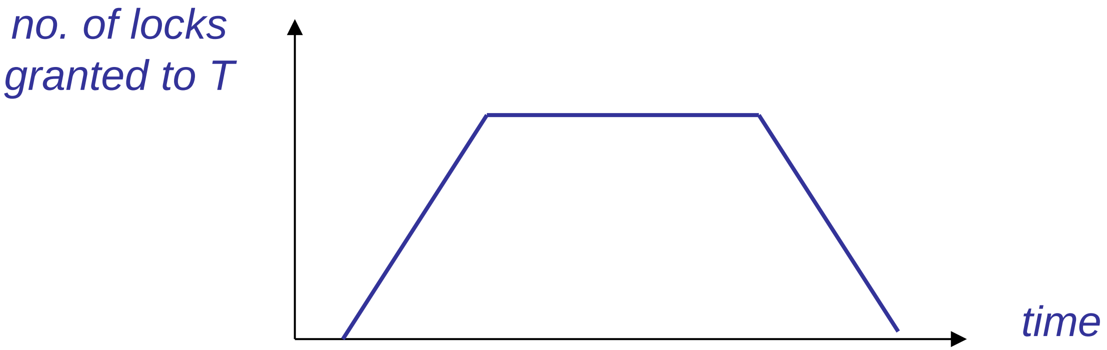


Properties of Strict 2PL protocol

- Two schedules are said to be conflict equivalent if:
  - They involve the same transactions
  - They order every pair of conflicting actions of two committed transactions in the same way
- A schedule is said conflict serializable if it is conflict equivalent to some serial schedule
- Every conflict serializable schedule is serializable
  - If data cannot be added or deleted (only modified)
- A schedule is conflict serializable if and only if its serializability graph is acyclic
- Strict 2PL protocol generates only acyclic graphs

2-phase lock (2PL) protocol

- With respect to Strict 2PL protocol, the second rule is now:
  - A transaction cannot request additional locks once it releases any lock
- Thus, every transaction has two phases:
  - Growing: locks are acquired
  - Shrinking: locks are released



Properties of 2PL protocol

- Even 2PL protocol ensures acyclicity of the serializability graph, therefore allowing only serializable schedules
- Schedules generated by the 2PL protocol are not strict
- The Strict 2PL protocol only generates strict schedules
    - Therefore, Strict 2PL also avoids cascading aborts
- Usually, DBMSs use Strict 2PL

Preventing phantom row

- Among the considered problems, is the most difficult to solve
- Existing solutions are quite different with respect to complexity and concurrency level allowable:
  - A S-lock can be acquired on the whole table, then X-lock are requested only for tuples to be modified
  - A new lock type is introduced, called "predicate lock", concerning all tuples satisfying a given predicate (location = 'Bologna' in the example)
  - If an index on location exists, a lock is requested on the leaf containing 'Bologna'
- In a DBMS, the situation is actually more complex
  - both for the different lock types
  - and for the "granularity" allowed for locks (at the level of attribute, tuple, page, table, …)

Lock management

- The Lock Manager maintains
  - A table containing active transactions
    - Also includes a list of locks held by any transaction
  - A lock table describing, for each data “object”, the type of the lock
    - Data “objects” can be pages, records, tree nodes, etc.
- Every lock table entry contains:
  - The type of lock (S/X)
  - The number of transactions currently holding the lock
  - A queue of lock requests

Implementing the lock protocol (i)

- A transaction can request a lock on a data “object” specifying the type of lock
  - If an S-lock is requested, the request queue is empty and the object is not currently X-locked, the lock is granted and the lock table is updated
  - If an X-lock is requested and no transaction currently holds a lock on the object (thus the request queue is also empty), the lock is granted and the lock table is updated
  - Otherwise, the request cannot be granted, the lock request is queued and the transaction is suspended

Implementing the lock protocol (ii)

- At the end of each transaction (COMMIT or ABORT)
  - All its locks are released
  - For each lock entry, the lock request at the head of the queue is examined and, in case, the transaction is woken up and given the lock
  - Only the head of the queue is examined, in order to avoid "starvation" of transactions
- Clearly, the implementation of lock/unlock commands must ensure they are atomic operations
  - A synchronization mechanism should therefore be implemented to allow concurrent access to the lock table (e.g., semaphores)

Deadlock management

- Protocols for deadlock management are similar to those you might have seen in other courses
  - Prevention techniques
  - Detection techniques
- In both cases, it is required that a transaction is aborted
- Usually, detection techniques are preferred,
  since deadlocks are rather infrequent

Deadlock prevention

- Every transaction is assigned a priority
- Let us suppose that T1 requests a lock on O and T2 has on O a conflicting lock
  - If T1 has priority > than T2 is allowed to wait, otherwise T1 aborts (wait-die)
  - If T1 has priority > than T2, T2 is aborted, otherwise T1 waits (wound-wait)
- In both cases, deadlocks cannot occur
- Typically, priority is given by the timestamp
  - Whenever a transaction is aborted and re-created,
    it is given the original timestamp to avoid “starvation”

Comparison wait-die/wound-wait

- wait-die is non-preemptive: T can be aborted only because it requests a lock, not due to requests of other transactions
- With wait-die:
  - An “old” transaction it tends to wait for younger transactions (and to “grow older”)
  - A “young” transaction may be repeatedly aborted
  - A transaction holding all the lock it needs will never be aborted

Conservative 2-phase lock

- A transaction requests all the lock it needs at the beginning
  - Clearly, deadlocks cannot occur
  - Transactions are never put to sleep
- The drawback is that locks are kept for a longer time, if there are only a few requests
- On the other hand, when multiple requests are present, transactions are never put on hold, thus locks are kept (on average) for a shorter time

Deadlock detection

- Deadlocks are rather infrequent and usually involve only a few transactions
- The control is performed periodically and uses the waits-for graph
  - A node for each active transaction
  - An arc from a waiting transaction to the transaction currently holding the lock
- A deadlock correspond to a cycle in the waits-for graph

Choosing the transaction to abort

- In order to solve a deadlock situation,
  we should select a transaction in a cycle and abort it
- Criteria:
  - Least work done
  - Most work yet to be done
  - Least number of times a transaction was aborted
  - Least number of locks held
  - …

Concurrency control in SQL

- Although the DBMS is responsible for managing concurrent transactions (thus simplifying writing application code),
SQL provides two basic mechanisms to specify transaction-level behavior
- As said, lock requests are implicit, since requesting and managing locks is both time- and space-consuming,
if a transaction knows it has to process a large number of tuples in a relation, it can explicitly request a lock (SHARE or EXCLUSIVE MODE) on the whole table, e.g.:

```
LOCK TABLE Students IN SHARE MODE;
SELECT *
FROM Students
WHERE BirthDate<‘11/07/1982’;
```

Isolation levels

- The isolation level of an application controls the extent of protection for data used by that application, with respect to other application executing concurrently
- By choosing a possible isolation level, the user can obtain greater concurrency (and thus performance), at the cost of increasing the exposure to other applications' uncommitted changes

Isolation levels in SQL

- SQL standard defines 4 isolation levels
(we also report names used by DB2):

Isolation Level	DB2 terminology	Phantom	Unrepeatable Read	Dirty Read	Lost Update
Serializable	Repeatable Read (RR)	NO	NO	NO	NO
Repeatable Read	Read Stability (RS)	YES	NO	NO	NO
Read Committed	Cursor Stability (CS)	YES	YES	NO	NO
Uncommitted Read	Uncommitted Read (UR)	YES	YES	YES	NO

- In DB2, the default level is CS
- To change it (before connecting to the DB) the following SQL statement is used

```
CHANGE ISOLATION TO [RR|RS|CS|UR]
```

Isolation levels in DB2 (i)

- Repeatable Read
  - Locks all data used by the application
  - If a SELECT is executed on a table, the whole table is locked, not just the result tuples
- Read Stability
  - Locks all data retrieved by the application
  - If a SELECT is executed on a table,
    only the result tuples are locked

Isolation levels in DB2 (ii)

- Cursor Stability
  - Locks only the data currently used by the application
  - If a SELECT is executed on a table,
    only the current tuple is locked
- Uncommitted Read
  - The application can access uncommitted data
    from other applications
  - Useful if read-only tables or SELECT statements are only used

Advanced issues

- A DBMS should be able to manage concurrency at different granularity levels
- Concurrency control should be exerted also on index structures
- In a system with relatively light contention for data objects, the overhead for obtaining locks and following a lock protocol could be too high
    - Optimistic concurrency control (possible conflicts are checked at commit-time, possibly aborting transactions)
    - Timestamp-based concurrency control (transactions are ordered by way of their timestamp)

Durability control

- Until now, we only considered isolation of transactions
- As said, the Logging & Recovery Manager is in charge of atomicity and durability
- In practice, we should guarantee that all actions of committed transactions survive system crashes or media failures

The ARIES algorithms (Mohan & Rothermel, ’89)

- ARIES (Algorithms for Recovery and Isolation Exploiting Semantics) is a family of algorithms for locking, logging, and recovery for the management of persistent data
- Originally developed for System R, the original DBMS by IBM
- Currently adopted by several systems, among which:
  - DB2
  - SQL server
  - NT file system

Types of failures

- The different failures belong essentially to one of the following three classes:
  - Transaction failure: is the case of an aborted transaction
    - The effects of such transaction on the DB have to be un-done
  - System failure: the system has an hardware or software failure, stopping all current transactions, but the secondary memory (disks) is not damaged
  - Media/device failure: in this case the (persistent) content of the database is damaged
- Basic assumption:
  - Writing a page to disk is an atomic action

Buffer management

- When a transaction T modifies a page P, the Buffer Manager has 2 possible options:
  - No-steal policy: Keeping the page P within the buffer, waiting that T commits before writing it on disk
  - Steal policy: Writing P when it is “more convenient” (to free the buffer or to optimize I/O operations), possibly before T terminates
    - For efficiency reasons, DB2 uses the steal policy

Committing a Transaction

- When a transaction T commits, we have again two options:
  - Force policy: before "formalizing" the conclusion of the transaction, all pages modified by T are immediately forced to disk
  - No-force policy: the transaction is "formally" terminated; thus, some of its changes may still not have been written to disk
    - Again, for efficiency reasons, DB2 uses the no-force policy

Atomicity and Durability

- To handle failures, a DBMS exploits different mechanisms, in particular:
  - DataBase Dump: archive backup of (a part of) the DB
  - Log file (“trail” or “journal”): sequential file
    where all update actions executed by transactions are recorded

The Log

- A record is written on the log as a result of the following actions :
  - Update: update of the content of a page
  - Commit: correct termination of a transaction
  - Abort: incorrect termination of a transaction
  - End: termination of a transaction (following commit/abort)
  - Compensation: records the undoing of updates of a (failed) transaction

Update record

- The format of an update record for a transaction T modifying a DB page P is:

  (LSN, prevLSN, T, type, PID, before(P), after(P))

  - LSN: Log Sequence Number (record unique id)
  - prevLSN: LSN of the previous Log record concerning T
  - T: transaction unique id
  - type: record type (update in this case)
  - PID: modified page unique id
  - before(P): "before image" of P, that is, the content of P before the change
  - after(P): "after image" of P, that is, the content of P after the change

Compensation Record

- It is used when the change recorded in an update record is undone (e.g., because the transaction was aborted)
- The format of an update record for a transaction T is:

  (LSN, prevLSN, T, type, undoNextLSN, PID, before(P))

  - undoNextLSN represents the next record to be undone
    - If we are undoing record U, this corresponds to the prevLSN of U

Example of Log

LSN	prevLSN	T	type	PID	before(P)	after(P)
…						
235	-	T1	BEGIN			
236	-	T2	BEGIN			
237	235	T1	UPDATE	P15	(abc, 10)	(abc, 20)
238	236	T2	UPDATE	P18	(def, 13)	(ghf, 13)
239	237	T1	COMMIT			
240	239	T1	END			
241	238	T2	UPDATE	P19	(def, 15)	(ghf, 15)
242	-	T3	BEGIN			
243	241	T2	UPDATE	P19	(ghf, 15)	(ghf, 17)
244	242	T3	UPDATE	P15	(abc, 20)	(abc, 30)
245	243	T2	ABORT			
246	244	T3	COMMIT			
247	243	T2	END			
…						

WAL Protocol

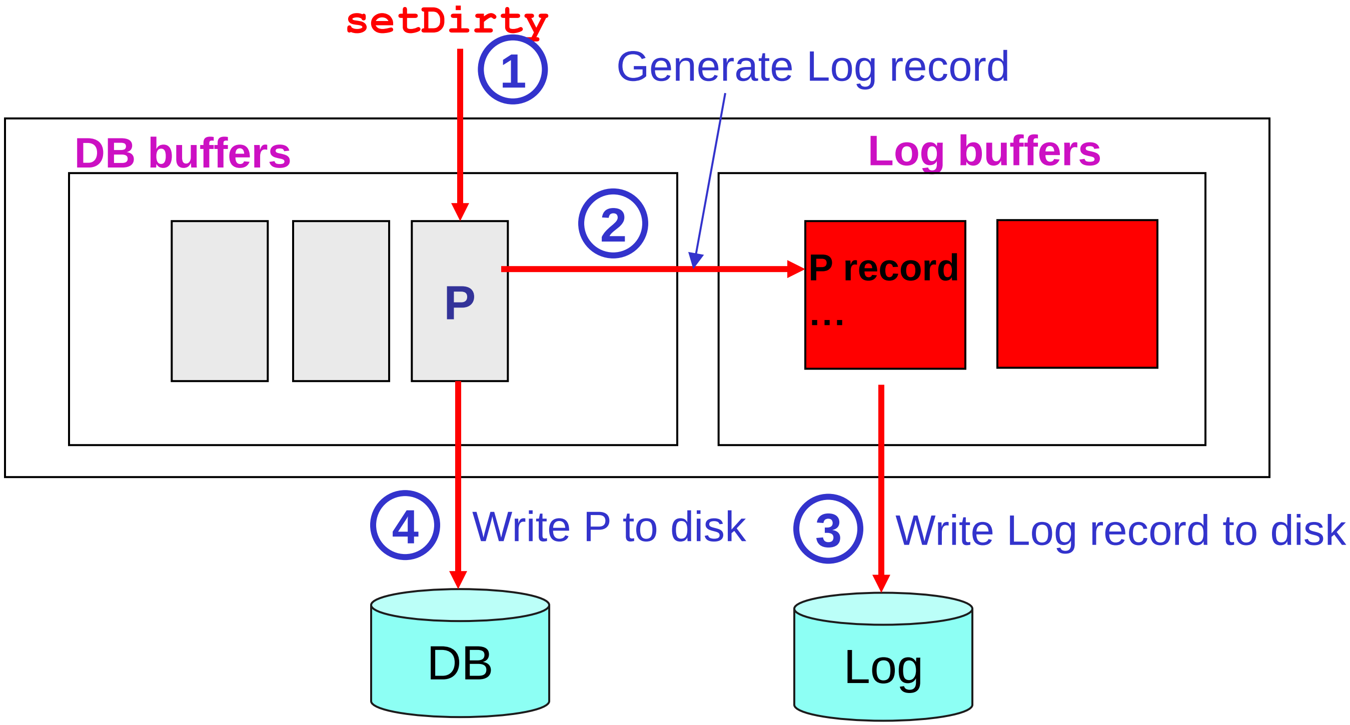
- In order to use the Log to restore the DB state after a failure, it is fundamental to apply the so-called WAL protocol ("Write-ahead Logging"):

before writing a page P to disk, every update record describing a change to P should be written to the Log

- Intuitively, if the WAL protocol is not observed, it is possible that:
  - A transaction T modifies the DB updating a page P
  - A system failure occurs before the corresponding update record has been written to the Log
- In this case, it is evident that there would be no way to restore the DB to its initial state

Implementing the WAL protocol

- The Buffer Manager is responsible for ensuring compliance with the WAL protocol
  - To this end, the Buffer Manager handles both the DB and the Log buffers
- In figure, we report the order in which the different operations related to the modification of a page are performed



Implementing the Log

- The Log must be written on stable storage
  - Clearly, the Log should survive both system and media failures
  - Stable storage is achieved by maintaining multiple copies (perhaps in different locations) of the Log in different permanent devices (disks/tapes)
- The Log allows the Recovery Manager
  - to undo actions of aborted or incomplete transactions and
  - to redo actions of committed transactions
- A transaction can be considered committed

  only when its log records have been written on stable storage!

Transaction failure

- With the steal policy, if a transaction T aborts it is possible that some of the pages it changed have been already written on disk
- To undo such changes (UNDO), we scan the Log backwards (using the prevLSN field) and restore on the DB the “before images” of pages modified by T

LSN	prevLSN	T	type	PID	before(P)	after(P)
…						
236	-	T2	BEGIN			
237	235	T1	UPDATE	P15	(abc, 10)	(abc, 20)
238	236	T2	UPDATE	P18	(def, 13)	(ghf, 13)
239	237	T1	COMMIT			
240	238	T2	UPDATE	P19	(def, 15)	(ghf, 15)
241	-	T3	BEGIN			
242	240	T2	UPDATE	P19	(ghf, 15)	(ghf, 17)
243	241	T3	UPDATE	P15	(abc, 20)	(abc, 30)
244	242	T2	ABORT			

System failure

- With a system failure, all transactions whose COMMIT cannot be found in the Log have to be undone
- If the no-force policy is adopted, it could be the case that some changes made by a committed transaction T have been not written to disk
  - Therefore, T has to be re-done, rewriting the “after images” found in the Log

LSN	prevLSN	T	type	PID	before(P)	after(P)
…						
235	-	T1	BEGIN			
236	-	T2	BEGIN			
237	235	T1	UPDATE	P15	(abc, 10)	(abc, 20)
238	236	T2	UPDATE	P18	(def, 13)	(ghf, 13)
239	237	T1	COMMIT			
…						

How to avoid useless page writes

- In order to avoid rewriting all “after images” of pages modified bi committed transactions, the Buffer Manager adopts the following technique:
  - When a page P is updated by a transaction T, the corresponding log record is generated with a given LSN
  - Such LSN is written in the page header of P

Page P15 on disk

Page Header	PID	LSN
	P15	293

LSN	prevLSN	T	type	PID	before(P)	after(P)
…						
237	…	T1	UPDATE	P15	(abc, 10)	(abc, 20)
238	…	T2	UPDATE	P15	(abc, 20)	(ghf, 13)
…						
327	…	T3		P15	(ghf, 13)	(ghf, 18)
…						

- When T is re-done and we find a log record concerning P with LSN = k, if LSN(P)≥ k there is no need to re-write P
- We read all pages updated by T, but we only write the ones that have not been already updated

Checkpoint

- As we will see, the restart procedure has the goal of restoring the DB to a consistent state after a system failure
- In order to reduce the amount of work needed during restart, a “checkpoint” is performed periodically, by forcing updated pages to disk
  - Checkpoint execution is recorded by writing on the Log a CKP (checkpoint) record including the transaction table and the dirty pages table
  - In this way, if T has been committed before checkpoint, T needs not to be re-done

LSN	prevLSN	T	type	PID	before(P)	after(P)
237	…	T3	UPDATE	P15	…	…
238	…	T2	UPDATE	P18	…	…
239	…	T1	UPDATE	P17	…	…
240	…	T1	COMMIT			
241	…	T2	COMMIT			
242			CKP			
243	…	T3	UPDATE	P19	…	…

Using ARIES

- ARIES allows the use of steal, no-force policies
- The DBMS restart after a crash is performed by the Recovery Manager in three steps:
  - Analysis: it determines (a conservative superset of) dirty pages and transactions that were active at the time of the crash
  - Redo: redoes all actions, starting from a particular point in the log
  - Undo: undoes all actions of aborted transactions

Basic principles of ARIES

- WAL protocol
- Repeating history by way of REDO
- Logging updates during UNDO
  - This avoids repeating UNDO multiple times in case of repeated failure/restart sequences

ARIES: system failure (i)

- First of all, the analysis phase is performed
- In the REDO phase, all updates to the pages which were dirty at the moment of the crash are re-done (forwards)
- In the UNDO phase, all updates of transactions which were active at the moment of the crash are un-done (backwards)

ARIES: system failure (ii)

- Analysis phase:
  - Search the most recent checkpoint (backwards)
  - Restore the corresponding transaction and dirty pages tables
  - Analyze the log forwards:
    - If a transaction terminates (ABORT/COMMIT record) it is removed from the transaction table
    - Every new transaction (BEGIN record) is added to the transaction table
    - Every page which is dirtied by an update or compensation record is added to the dirty pages table
  - The transaction table only contains the transactions which were active at the moment of the crash

ARIES: system failure (iii)

- REDO phase:
  - We start with the record having the least LSN, considering those included in the dirty pages table built during the analysis phase
    - It is possible that such record precedes the last CKP
  - We continue examining the log (forwards) re-doing all update and compensation records, unless:
    - The corresponding page is not in the dirty pages table
    - The LSN value in the page is ≥ the record LSN
  - The LSN value in the page is updated

ARIES: system failure (iv)

- UNDO phase:
  - We identify all those transactions which were active at the time of the crash
  - All actions of such transactions are un-done (backwards)
  - We start with the transaction with the most recent (highest) LSN
    - If the record is a compensation record, we proceed to the previous record (undoNextLSN field), unless this is 0; in such case, the transaction has been completely undone
    - If the record is an update record, we perform the undo, write a compensation record, and proceed to the previous record (prevLSN field)

ARIES: crash during restart

- Writing compensation records allows handling repeated system crashes, in particular in the restart procedure (during UNDO)
- In fact, compensation records specify that such updates have been already undone during a previous UNDO phase
- If the crash happens during the analysis phase, this should be restarted from the beginning
- If the crash happens during the REDO phase, this should be restarted from the beginning
  - Possibly, some pages are not rewritten during the new REDO

Media failure

- In case of a media failure we have to restore a copy of the DB (DataBase Dump)
- The DB dump is similar to a checkpoint
- At the restart, after restoring the dump, the “regular” recovery procedure is applied
  - All committed transactions are redone
  - All un-committed transactions are undone

Other algorithms for durability (i)

- If the Buffer Manager uses the no-steal policy, UNDO is not needed
- If the Buffer Manager uses the force policy, REDO is not needed
- Why ARIES is the most commonly used algorithms?
  - Because it favors the normal operation of the DBMS, under the hypothesis that failures are infrequent
  - In fact, other algorithms (greatly) complicate transaction management

Other algorithms for durability (ii)

- UNDO/no-REDO
  - Updates of a transaction T are written in stable memory before T terminates
- no-UNDO/REDO
  - Updates of a transaction T are written in stable memory after T terminates
- no-UNDO/no-REDO
  - Updates of a transaction T are written in stable memory when T terminates (as an atomic action)

Example (i)

- Suppose the log contains the following records:

LSN	prevLSN	T	type	PID	before(P)	after(P)
1	-	T1	BEGIN			
2	-	T2	BEGIN			
3	1	T1	UPDATE	PA	ValA	ValA’
4	2	T2	UPDATE	PB	ValB	ValB’
5	4	T2	UPDATE	PA	ValA’	ValA”
6	3	T1	UPDATE	PC	ValC	ValC’
7	5	T2	COMMIT			

- Transaction T1 modifies page PD
- A crash occurs before the corresponding log record is written
  - PB written; PA, PC, PD not

Example (ii)

- Analysis phase:
  - T1 added to transaction table
  - PA added to dirty pages table (LSN=3)
  - T2 added to transaction table
  - PB added to dirty pages table (LSN=4)
  - PC added to dirty pages table (LSN=6)
  - T2 deleted from transaction table
  - PD not added (WAL!)

LSN	prevLSN	T	type	PID	before(P)	after(P)
1	-	T1	BEGIN			
2	-	T2	BEGIN			
3	1	T1	UPDATE	PA	ValA	ValA’
4	2	T2	UPDATE	PB	ValB	ValB’
5	4	T2	UPDATE	PA	ValA’	ValA”
6	3	T1	UPDATE	PC	ValC	ValC’
7	5	T2	COMMIT			

Example (iii)

- REDO phase:
  - PA read from disk
    - pageLSN=0<LSN=3: action is redone, pageLSN=3
  - PB read from disk
    - pageLSN=4≥LSN=4: action is not redone
  - PA read from disk
    - pageLSN=3<LSN=5: action is redone, pageLSN=5
  - PC read from disk
    - pageLSN=0<LSN=6: action is redone, pageLSN=6
  - A END record is added for T2

LSN	prevLSN	T	type	PID	before(P)	after(P)
1	-	T1	BEGIN			
2	-	T2	BEGIN			
3	1	T1	UPDATE	PA	ValA	ValA’
4	2	T2	UPDATE	PB	ValB	ValB’
5	4	T2	UPDATE	PA	ValA’	ValA”
6	3	T1	UPDATE	PC	ValC	ValC’
7	5	T2	COMMIT			

Example (iv)

- UNDO phase:
  - T1 is the only active transaction
  - PC is restored to ValC
    - A compensation record is written with undoNextLSN=3
  - PA is restored to ValA
    - A compensation record is written with undoNextLSN=0
  - A END record is added for T1
  - Note that the value of PA updated by T2 has been overwritten (with Strict 2PL this would not happen)

LSN	prevLSN	T	type	PID	before(P)	after(P)
1	-	T1	BEGIN			
2	-	T2	BEGIN			
3	1	T1	UPDATE	PA	ValA	ValA’
4	2	T2	UPDATE	PB	ValB	ValB’
5	4	T2	UPDATE	PA	ValA’	ValA”
6	3	T1	UPDATE	PC	ValC	ValC’
7	5	T2	COMMIT			

Example: crash during restart (i)

- Suppose the log contains the following records:

LSN	prevLSN	T	type	PID	before(P)	after(P)
1	-	T1	BEGIN			
2	-	T2	BEGIN			
3	1	T1	UPDATE	PA	ValA	ValA’
4	2	T2	UPDATE	PB	ValB	ValB’
5	3	T1	ABORT			
6	5	T1	COMP	PA	ValA	undoNextLSN: -
7	-	T3	BEGIN			
8	7	T3	UPDATE	PC	ValC	ValC’
9	4	T2	UPDATE	PA	ValA	ValA”

- A system crash occurs
  - The ABORT of T1 is handled as a “regular” UNDO

Example: crash during restart (ii)

- Analysis phase:
  - PA (LSN=3), PB (LSN=4), and PC (LSN=8) added to dirty pages table
  - T2 and T3 added to transaction table (T1 added and deleted)
- REDO phase:
  - LSN=3 is the first record to be redone

LSN	prevLSN	T	type	PID	before(P)	after(P)
1	-	T1	BEGIN			
2	-	T2	BEGIN			
3	1	T1	UPDATE	PA	ValA	ValA’
4	2	T2	UPDATE	PB	ValB	ValB’
5	3	T1	ABORT			
6	5	T1	COMP	PA	ValA	undoNextLSN: -
7	-	T3	BEGIN			
8	7	T3	UPDATE	PC	ValC	ValC’
9	4	T2	UPDATE	PA	ValA	ValA”

Example: crash during restart (iii)

LSN	prevLSN	T	type	PID	before(P)	after(P)
1	-	T1	BEGIN			
2	-	T2	BEGIN			
3	1	T1	UPDATE	PA	ValA	ValA’
4	2	T2	UPDATE	PB	ValB	ValB’
5	3	T1	ABORT			
6	5	T1	COMP	PA	ValA	undoNextLSN: -
7	-	T3	BEGIN			
8	7	T3	UPDATE	PC	ValC	ValC’
9	4	T2	UPDATE	PA	ValA	ValA”
CRASH, RESTART						
10	9	T2	COMP	PA	ValA	undoNextLSN: 4
11	8	T3	UPDATE	PC	ValC	undoNextLSN: -
12	11	T3	END			

- UNDO phase:
  - Records to be undone are
    - LSN=9 for T2
    - LSN=8 for T3
  - PA is restored to ValA
    - A compensation record is written with undoNextLSN=4
  - PC is restored to ValC
    - A compensation record is written with undoNextLSN=0
  - New crash!

Example: crash during restart (iv)

- Analysis phase:
  - PA (LSN=3), PB (LSN=4), and PC (LSN=8) added to dirty pages table
  - T2 added to transaction table (T1 and T3 added and deleted)

LSN	prevLSN	T	type	PID	before(P)	after(P)
1	-	T1	BEGIN			
2	-	T2	BEGIN			
3	1	T1	UPDATE	PA	ValA	ValA’
4	2	T2	UPDATE	PB	ValB	ValB’
5	3	T1	ABORT			
6	5	T1	COMP	PA	ValA	undoNextLSN: -
7	-	T3	BEGIN			
8	7	T3	UPDATE	PC	ValC	ValC’
9	4	T2	UPDATE	PA	ValA	ValA”
CRASH, RESTART						
10	9	T2	COMP	PA	ValA	undoNextLSN: 4
11	8	T3	UPDATE	PC	ValC	undoNextLSN: -
12	11	T3	END			

Example: crash during restart (v)

- REDO phase:
  - LSN=3 is the first record to be redone, LSN=11 the last one
  - If some of the pages have already been written on disk, they are not re-written (their LSN is up-to-date)

LSN	prevLSN	T	type	PID	before(P)	after(P)
1	-	T1	BEGIN			
2	-	T2	BEGIN			
3	1	T1	UPDATE	PA	ValA	ValA’
4	2	T2	UPDATE	PB	ValB	ValB’
5	3	T1	ABORT			
6	5	T1	COMP	PA	ValA	undoNextLSN: -
7	-	T3	BEGIN			
8	7	T3	UPDATE	PC	ValC	ValC’
9	4	T2	UPDATE	PA	ValA	ValA”
CRASH, RESTART						
10	9	T2	COMP	PA	ValA	undoNextLSN: 4
11	8	T3	UPDATE	PC	ValC	undoNextLSN: -
12	11	T3	END			

Example: crash during restart (vi)

- UNDO phase:
  - The only record to be undone is LSN=4 for T2
  - PB is restored to ValB
    - A compensation record is written with no undoNextLSN

LSN	prevLSN	T	type	PID	before(P)	after(P)
1	-	T1	BEGIN			
2	-	T2	BEGIN			
3	1	T1	UPDATE	PA	ValA	ValA’
4	2	T2	UPDATE	PB	ValB	ValB’
5	3	T1	ABORT			
6	5	T1	COMP	PA	ValA	undoNextLSN: -
7	-	T3	BEGIN			
8	7	T3	UPDATE	PC	ValC	ValC’
9	4	T2	UPDATE	PA	ValA	ValA”
CRASH, RESTART						
10	9	T2	COMP	PA	ValA	undoNextLSN: 4
11	8	T3	UPDATE	PC	ValC	undoNextLSN: -
12	11	T3	END			
CRASH, RESTART						
13	10	T2	COMP	PB	ValB	undoNextLSN: -
14	13	T2	END			