14. Transactions

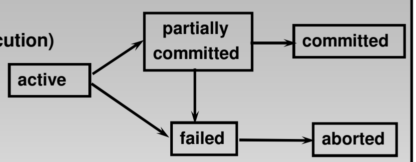
- Transaction: unit of program execution
  - Begin
  - Commands i.e. retrieves and updates
  - Commit (end of transaction)
- Atomicity: all or nothing gets done
- Consistency: preserves consistency of the database
- Isolation: unaware of other concurrent transactions (as if none)
- Durability: after completion the results are permanent even at system crashes

- Potential ACID violations
  - consistency is the responsibility of the programmer
    - we assume that all Trans are correct and not malicious
  - if the system crashes at half way through➔ atomicity violation
  - if another trans modifies the data while executing ➔ isolation violation
  - if the transfer of 50 from A to B is lost after commit ➔ durability -”-

```
T1: begin trans
read(A);
A:=A - 50;
write(A);
read(B).;
B:=B + 50;
write(B);
commit
```


Transactions

- transaction states:
  - active (executing)
  - partially committed (after last statement’s execution)
  - failed (if can no longer proceed)
  - aborted (after trans has been rolled back)
  - committed (after successful completion)

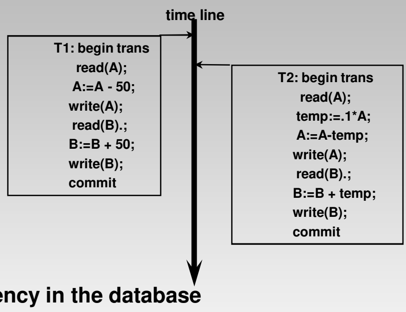


- An aborted transaction can be
  - restarted as a new transaction
  - killed if it is clear that it will fail again

- roll-back
  - can be requested by the transaction itself (go back to a given execution state)
  - some actions are not rollbackable (e.g. a printed message, or ATM cash withdrawal)

Transactions

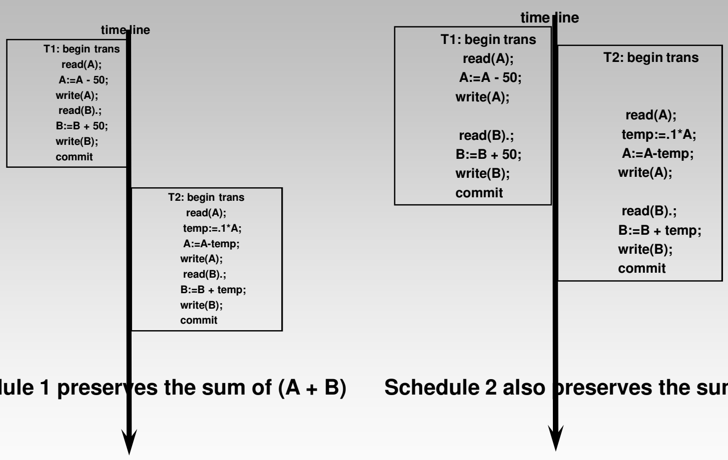
- in a multiprogramming environment several transactions may be executed concurrently to increase ***throughput***
- T1 and T2 are ***interleaved***



- Serial schedules guarantee consistency in the database
  - A serial schedule is equivalent to having the transactions execute one-at-a-time
    - Number of serial schedules = n! (10!=3.6M 15!=1.3T)
  - An interleaved (non-serial) schedule permits concurrent execution
    - Number of interleaved schedules >> n!

357

Serial and Interleaved Schedules



Schedule 1 preserves the sum of (A + B)

Schedule 2 also preserves the sum of (A + B)

Schedule 1

Schedule 2

358

–2

Schedule 4

Schedule 4 does not preserve the sum of (A + B)

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Serializability

- **Basic Assumption – Each transaction preserves database consistency.**
- **Thus any serial execution of a set of transactions preserves database consistency**
  - **T1 followed by T2**
  - **T2 followed by T1**
- **Both are serial executions, therefore, both preserve consistency (correctness) even when the database differs**

Serializability

- An interleaved schedule is *serializable* if it is equivalent to a serial schedule.
- We focus on read and write operations because these are the ones that can violate ACID properties

Transaction Conflicts

- Conflict: two operations on the same data item by different transactions are in **conflict** if at least one of the operations is a write
- if two consecutive operations in a schedule S are not in conflict, then we swap the two to produce another schedule S' that is conflict-equivalent with S

Example: read(B) and write(B) on T0 do not conflict with read(A) and read(B) on T1 and, therefore, can be moved all the way up.

T0	T1
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

$\Longrightarrow$

T0	T1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

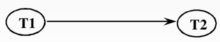
- schedule S is **conflict serializable** if it is conflict-equivalent to a serial schedule

Testing for Conflict Serializability

- precedence graph: a node per transaction; an edge Ti ➔ Tj if
  - Ti does a write(Q) before Tj a read(Q)
  - Ti does a read(Q) before Tj a write(Q)
  - Ti does a write(Q) before Tj a write(Q)
- If the precedence graph has a cycle, the schedule is not conflict serializable
  - check for cycles in a graph is O(n^2) [more precisely O(max(m,n)) m edges n nodes]

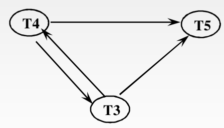
Schedule 2:

T1	T2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)



Schedule 3:

T4	T3	T5
read(Q)		
	write(Q)	
write(Q)		
		write(Q)



A Not Conflict-Serializable may be Serializable

- Precedence graph for Schedule 4 has a cycle, therefore it is not conflict-serializable

- Still the execution of this interleaved protocol is equivalent to a serial schedule T1,T2 (also T2,T1)
- Therefore, it is serializable
- In order to decide if a not conflict-serializable is serializable
  - we need to look into the reads and writes inside the transactions
  - It is too expensive to analyze all these reads and writes inside each transaction

Schedule 4:

T1	T2
read(A)	
A:=A-50	
write(A)	
	read(B)
	B:=B-10
	write(B)
read(B)	
B:=B+50	
write(B)	
	read(A)
	A:=A+10
	write(A)

Recoverability

- Need to address the effect of transaction failures on other running transactions.

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- If T8 aborts, T9 must have been aborted (it may have given a wrong data to a user).

- **Dirty read**: Reading a value written by an uncommitted transaction

  T_9 above has a dirty read

- **Recoverable schedule** — if a transaction Tj reads a data item previously written by a transaction Ti , Ti must commit before Tj

  Otherwise, the schedule is not recoverable

Cascading Rollbacks

- *Cascading rollback:* a single transaction failure leads to a series of transaction rollbacks.
- Example:

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

none of the transactions has yet committed (so the schedule is recoverable)

- If T_{10} fails, T_{11} and T_{12} must also be rolled back.
- Can lead to the undoing of a significant amount of work
- *Cascadeless schedules:* avoid cascading rollbacks (desirable)
- Every cascadeless schedule is also recoverable

Concurrency Control Protocol

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict serializable
  - recoverable, and
  - preferably cascadeless
- Executing one transaction at a time results serial schedules that are
  - recoverable
  - cascadeless, but
  - provides a poor degree of concurrency
- Testing a schedule for serializability *after* it has executed is a little too late!
- Goal – to develop concurrency control protocols that will assure serializability

367

Weak Levels of Consistency

- For some applications we are willing to live with weak levels of consistency
- Allow schedules that are not serializable
  - E.g. a read-only transaction that wants to get an approximate average of student grades in CS
  - E.g. database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

368

Levels of Consistency in SQL

- *Serializability*: is the default
- Non-serializable executions:
  - Repeatable reads: allows only committed records to be read, and repeating a read should return the same value
    - Suffers from the phantom phenomenon

```
T1:  select avg(sal) from emp where emp.dno='toy'
T2:  { insert into emp values(123,82K,toy)
       insert into emp values(124,75K,toy) }
```

T1 may or may not see some of the records inserted by T2

  - Read committed — only committed records can be read, but successive reads of record may return different committed values.
  - Read uncommitted allows even uncommitted data to be read

369

Transaction Definition in SQL

- By default, in SQL every statement also commits implicitly if it executes successfully
- Implicit commit can be turned off by a database command
  - Sqlplus: begin transaction <SQL-statements> ...
  - JDBC: connection.setAutoCommit(false);
- A transaction in SQL ends by:
  - The commit transaction keyword commits current transaction and begins a new one
  - rollback transaction causes current transaction to abort

370

15. Concurrency Control
Lock-Based Protocols (pessimistic)

- ***locks***:
  - **S**hared for reads – can have multiple reads
  - e**X**clussive for writes - only one write at a time

Compatibility Matrix

	S	X
S	Yes	No
X	No	No

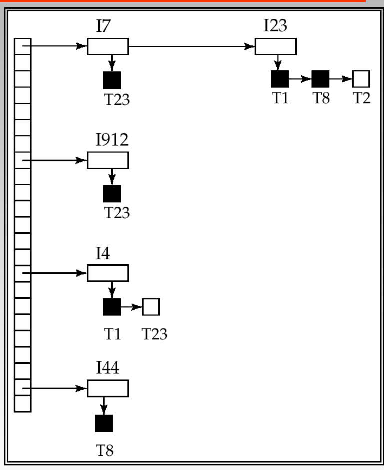
- ***Locking Protocol:***
  rules for placing and releasing locks by transactions
  - Get an S before a read (minimum)
  - Get an X before a write
  - Release locks whenever

time line

T1: begin trans	T2: begin trans
lock-S(A)	
read(A);	
lock-X(A);	lock-S(A);
A:=A - 50;	read(A);
write(A);	temp:=.1*A;
unlock(A);	A:=A-temp;
	lock-X(A);
lock-S(B);	write(A);
read(B).;	unlock(A);
B:=B + 50;	lock-X(B);
lock-X(B);	read(B).;
write(B);	B:=B + temp;
unlock(B);	write(B);
commit	unlock(B);
	commit

Lock Manager & Lock Table

- Lock manager is a process
  - Receives lock / unlock requests (Ti,Item,Lock-type)
  - Sends back lock grants or aborts on deadlocks
- Lock table stores granted / queued requests per locked data item and transaction making the request
  - Black rectangles indicate granted locks, white ones indicate waiting requests
  - A New request is added to the end of the queue if the item exists and granted if compatible lock
  - If the item is not in the table, a new entry is created and requested lock is granted
- Unlock requests
  - Remove the item-transaction entry followed queued requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager keep a list of locks held by each transaction, to find them efficiently



Pitfalls of Lock-Based Protocols Deadlocks

- Consider the schedule

T_3	T_4
lock-x (B)	
read (B)	
$B := B - 50$	
write (B)	
	lock-s (A)
	read (A)
	lock-s (B)
lock-x (A)	

- executing lock-S*(B)* causes T_4 to wait for T_3 to release its lock on *B*
- executing lock-X*(A)* causes T_3 to wait for T_4 to release its lock on *A*
- Neither T3 nor T4 can make progress —
- Such a situation is called a ***deadlock***.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released
- Repeated roll backs of the same transaction due to deadlocks may lead to ***starvation***
- 95% of deadlocks are between 2 transactions

Deadlock Prevention

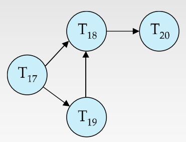
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state.
- One simple prevention strategy is:
  - Require that each transaction locks all its data items before it begins execution (predeclaration). USELESS- you do not know in advance
- Timeout-Based Schemes:
  - a transaction waits for a lock only for a specified amount of time. After that, the transaction is rolled back.
    - No deadlocks are possible
  - simple to implement
    - Starvation is possible
  - difficult to determine good timeout value

More Deadlock Prevention Strategies

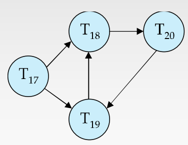
- use transaction timestamps for deadlock prevention
- ***wait-die*** scheme — non-preemptive
  - An older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back immediately
  - a transaction may die several times before acquiring needed data item
- ***wound-wait*** scheme — preemptive
  - An older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions wait for older ones.
  - may have fewer rollbacks than *wait-die* scheme
- In both wait-die and in wound-wait schemes, a rolled back transactions is restarted with its original timestamp
  - Older transactions thus have precedence over newer ones, and starvation is hence avoided.

Deadlock Detection

- Instead of preventing, let deadlocks occur and from time to time detect them
- Deadlocks can be described as a ***wait-for graph***, $G = (V,E)$,
  - V is a set of vertices (all the transactions running in the system)
  - E is a set of edges: $T_i \rightarrow T_j$ meaning T_i waits for a lock held by T_j
- When T_i requests a data item currently being held by T_j, then we add $T_i \rightarrow T_j$ in the wait-for graph
- When T_j releases a lock held on a data item that Tj is waiting for, we remove Ti → Tj
- The system is in a deadlock state iff the wait-for graph has a cycle



Wait-for graph without a cycle



Wait-for graph with a cycle

- Deadlock-detection algorithm runs periodically to look for cycles.

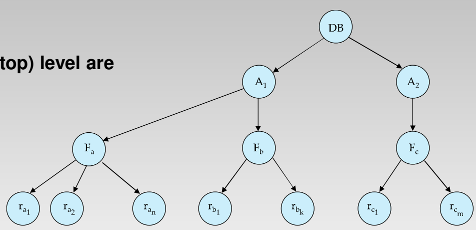
Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to be rolled back (made a victim) to break deadlock.
  - Select as victim that will incur minimum cost.
  - Rollback --
    - *Total rollback*: abort the transaction and then restart it.
    - *Partial rollback*: roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Multiple Granularity of Locking

- Single level locks have high overhead (too many items to lock or unlock)
- Solution: Allow locking of various size data objects **explicitly** and have its components locked **implicitly**
  - hierarchy of data granularities

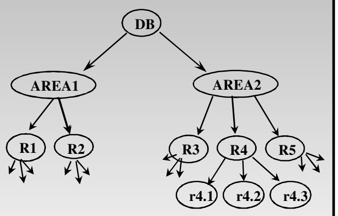
  The levels, starting from the coarsest (top) level are
  - database
  - area
  - file
  - record



- When a transaction locks a node in the tree ***explicitly***, it ***implicitly*** locks all the node's descendents in the same mode.
- Hierarchical locking controls overhead
  - fine granularity (lower in the tree): higher concurrency but higher locking overhead
  - coarse granularity (higher in the tree): lower locking overhead but also lower concurrency

Hierarchical Locking Prorocol

- places locks at the right level but always traverses the tree from the root
  - how does a transaction T at a node knows that the lock it wants is compatible with locks below the level are compatible?
  - search all levels below? High overhead-defeats the purpose.
  - need a quick method for knowing if an incompatible lock is in its descendants
- Introduce *intention locks* which make implicit locks to the descendants:
  - a transaction T with an IS (IX) lock at a node can later place an explicit S (X) on its descendants (no implicit or explicit S (X) locks on them); contrast this with S (X) that places an implicit S (X) lock to the whole subtree)
  - a transaction T with an SIX implies an implicit S lock below but permits ONLY transaction T to make later an explicit requests for X, SIX, IX to any descendant.
  - only one transaction can hold an SIX on a node!
  - SIX is equivalent to S and IX at the same time



Hierarchical Locking Protocol

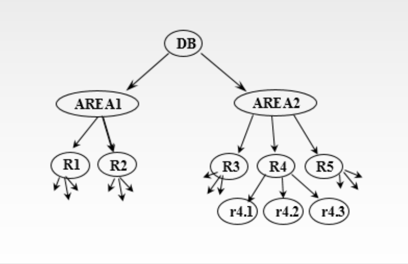
- compatibility matrix

	IS	IX	S	SIX	X
IS	Y	Y	Y	Y	N
IX	Y	Y	N	N	N
S	Y	N	Y	N	N
SIX	Y	N	N	N	N
X	N	N	N	N	N

- protocol: lock top-down; release bottom-up
  1. compatibility matrix must be observed
  2. the root of the tree must be locked first in the mode needed
  3. node Q can be locked by T in mode S or IS if parent(Q) is locked by T in IX or IS
  4. >> >> X, SIX, or IX >> >> IX or SIX
  5. T can lock a node if it has not unlocked any node (see 2-φ locking next)
  6. T can unlock a node Q only if none of the children of Q are locked by T
     - descendants are released first
- increases concurrency, reduces overhead, guarantees serializability

Hierarchical Locking Examples

- T1 wants to read tuple r4.3
  - lock DB, AREA2, and R4 in IS
  - lock r4.3 in S
- T2 wants to update tuple r4.2
  - lock DB, AREA2, R4 in IX mode
  - lock r4.2 in X
- T3 wants to read ALL records of R4
  - lock DB, AREA2 in IS and
  - lock R4 is S

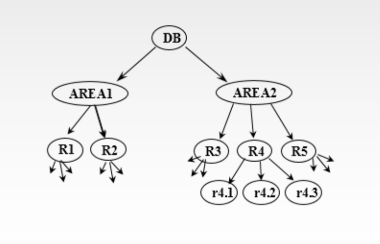


	IS	IX	S	SIX	X
IS	Y	Y	Y	Y	N
IX	Y	Y	N	N	N
S	Y	N	Y	N	N
SIX	Y	N	N	N	N
X	N	N	N	N	N

Hierarchical Locking Examples (cont.)

- R4 is S
- T4 wants to read the entire DB
  - lock DB in S
- T5 wants to read the entire relation R4 but update just a few tuples
  - lock DB, AREA2 in IX
  - lock R4 in SIX (permit to put X lock on descendants)
  - lock the updated tuples in X

	IS	IX	S	SIX	X
IS	Y	Y	Y	Y	N
IX	Y	Y	N	N	N
S	Y	N	Y	N	N
SIX	Y	N	N	N	N
X	N	N	N	N	N



Two-phase (2-φ) Locking

- *growing phase:* a transaction may only obtain locks (never release any locks)
- shrinking phase: a transaction may only release locks (never obtain new locks after the first release)
- *two-phase locking* guarantees serializability but no freedom of deadlocks
- *two-phase with lock conversion*:
  - S can be upgraded to X during the growing phase
  - X can downgraded to S during the shrinking phase

  2-φ idea: during the growing phase if T1 reads an item ewith S but not ready to update, it can keep it with S. This allows another T2 to read it and commit before T1 upgrades S to an X. T2 commits with S. When T1 is ready to write it, it upgrades S to an X instead of holding an X all along.
- Similarly, when T1 downgrades an X lock, other transactions can start reading it earlier.
- *strict two-phase locking* additionally requires that all X locks are held until commit time (this prevents anyone else seeing uncommitted data)
- *rigorous two-phase* locking requires that ALL locks are held until commit time
- most DBMSs implement either strict or rigorous 2-phase locking

383

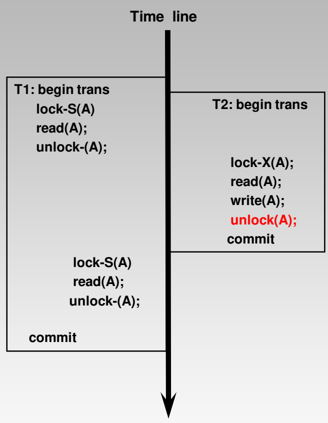
Degree 3 level Consistency ≡ Serializability

- Degree 3 is the maximum level of consistency
- Theorem: 2-φ locking guarantees serializability

384

Degree 2 level Consistency

- *Degree-two consistency:*
  - differs from 2-φ locking
  - S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - No repeatable reads
  - Serializability is not guaranteed



385

Variations of Degree 2 level Consistency

- *Repeatable read* (SQL-92)
  - only committed records to be read,
  - repeated reads of same record must return same value
  - not serializable – it may find some records inserted by a transaction but not find others (phantoms)
- *Read committed* (SQL-92)
  - only committed records can be read
  - successive reads of record may return different (but committed) values.
- *Cursor stability for iterators*
  - For reads, each tuple is S-locked, read, and the lock is immediately released
  - To modify a tuple it is X-locked and the X-lock is held till end of transaction
  - Special case of degree-two consistency
  - Dangerous only to be used in specialized situations where the programmer can guarantee consistency in non-serializable schedules

386

Degree 1 level of Consistency

- *Read Uncommitted* SQL-92
  - Allows uncommitted records to be read
  - Lowest level of consistency in SQL-92

16. Log-Based Recovery

- **Keep a log that stores all database update activities in the order they occur**

T-ID: transaction identifier
D-ID: data item identifier
Vold: old value of D-ID
Vnew: new value of D-ID

- ***6 types of transaction log records*:**
  - **<T,start>**
  - **Update log record of the form <T-ID, D-ID,V-old,V-new>**
  - **Redo-only record of the form <T-ID, D-ID, V-old>**
  - **<T,commit>**
  - **<T,abort>**
  - **<Checkpoint,L>**

Fragment of the log:
...<T35,start>,<T35,obj354,Boston,Detroit>,<T36,start>,<T36,obj653,45,65>,<T35,commit>, <T36,obj564,MA,MD><T37,start><T36 abort><T38,start> ...

Write-Ahead-Log (WAL)

a. log records are written to disk before database updates are written to disk (it's a must)

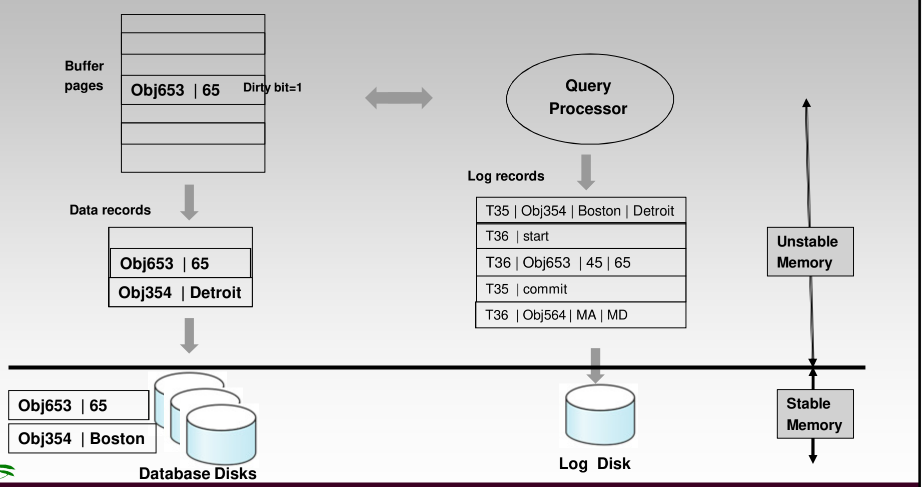
b. log records are batched before written to disk for efficiency

c. a transaction's <commit-T> record is not written to the log on disk before all update log records <T, D-ID,V-old,V-new> of T are written on disk (<commit-T> is written last on disk)

- when the log records are written to disk, changes to the DB can be applied too because even if *hell breaks loose*, they can be recovered from the log
- we use two primitives:
  - redo: applies a change to the database regardless of whether the change has been applied to disk or not
  - undo: applies a reverse action to the database regardless of whether the change has been applied on disk or not
  - multiple application of redos (undos) of the same action is equivalent to one- no harm if something that we redo (undo) is already on disk

Multibuffering

...<T35,start>,<T35,obj354,Boston,Detroit>,<T36,start>,<T36,obj653,45,65>,<T35,commit>,<T36,obj564,MA,MD>
<T37,start><T36 abort><T38,start>...<T38,commit> <checkpoint>...



During Normal Operation

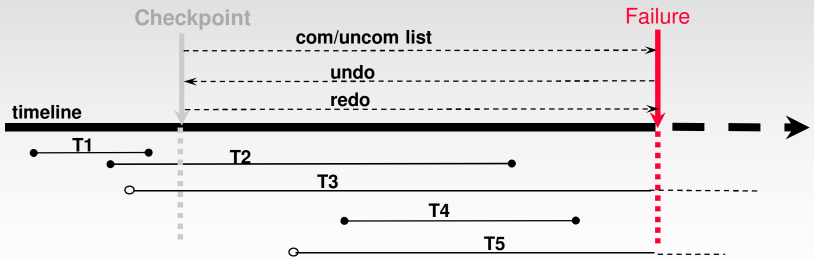
- Logging:
  - *<T-ID*.start> at transaction start
  - <T-ID, D-ID,V-old,V-new> for each update
  - <T-ID,commit> at transaction end

- Transaction rollback:
  - Let T-ID be the transaction to be rolled back
  - Scan log backwards from the end, and for each log record of <T-ID, D-ID,V-old, V-new>
    - perform the undo by writing V-old to D-ID
    - write a log record <T-ID, D-ID, V-old>

    such log records are called compensation log records
  - Once the record *<T-ID,* start> is found stop the scan and
    - write the log record <T-ID, abort>

Log-Based Recovery

- from checkpoint forward up to the failure to make the ***redo*** and ***undo*** lists for the committed and uncommitted transactions resp.
  - redo list: T2, T4
  - undo list: T3, T5
- from failure backwards and undo writes of uncommitted transactions
- from checkpoint forward and redo the writes of committed transactions (redos must be done AFTER the undos)
- expensive: 3 sequential scans of the active log



Frequency of Checkpointing

- often checkpointing speeds up recovery
  - log prior to checkpoint can be archived
  - if no checkpointing the log is incredibly long, has to be read sequentially 3 times-recovery will take for ever
- during checkpointing:
  - stop accepting new transactions and wait until all active transactions commit
  - output onto stable storage all log records in MM
  - output onto disk all modified (dirty) buffers
  - output onto stable storage a log record <checkpoint>

…<T35,start>,<T35,obj354,Boston,Detroit>,<T36,start>,<T36,obj653,45,65>,<T35,commit>,<T36,obj564,MA,MD><T37,start><T36 abort><T38,start>…<T38,commit> <checkpoint>…

- better checkpointing:
  - do not wait for the transactions to finish, but do not let them make updates to the buffers nor the update log
  - make the checkpoint log record to include the list L of active trans <checkpoint,L>
  - then on recovery, we have to go even further back from the checkpoint to find all the changes of all L transactions

…<T35,start>,<T35,obj354,Boston,Detroit>,<T36,start>,<T36,obj653,45,65>,<T35,commit>,<T36,obj564,MA,MD><T37,start><T36 abort><T38,start>…<T37,obj333,ABC,XYZ> <checkpoint,T37,T38>…

  - a more elaborate scheme that allows updates during recovery is called *fuzzy checkpointing*

393

Recovery Algorithm

- **Redo phase:**
  1. Find last <checkpoint *L*> record, and set undo-list to *L*.
  2. Scan log *forward* from above <checkpoint *L*> record
     1. Whenever a record <T-ID, D-ID,V-old,V-new> is found, redo it by writing V-new to D-ID
     2. Whenever a log record <T-ID,start> is found, add T-ID to undo-list
     3. Whenever a log record <T-ID,commit> *or* <T-ID,abort> is found, remove T-ID from undo-list

- **Undo phase:**
  1. Scan log *backwards* from end (crash)
     1. Whenever a log record <T-ID, D-ID,V-old,V-new> is found where *T-ID* is in undo-list perform same actions as for transaction rollback:
        1. perform undo by writing V-old to D-ID .
        2. write a log record <T-ID, D-ID,V-old>
     2. Whenever a log record <T-ID,start> is found where *T-ID* is in undo-list ,
        1. Write a log record <T-ID,abort>
        2. Remove T-ID from undo-list
     3. Stop when undo-list is empty

- After undo phase completes, normal transaction processing can commence

 394

Immediate vs. Deferred Modification

- Immediate or Any time Database Modification
  - writes of uncommitted transactions need to be undone (drawback)
  - writes of committed and uncommitted transactions have to be recorded to the log first
  - Benefit: buffers are emptied much sooner
- Deferred Database Modification
  - no writes to the database before transaction is partially committed- I.e. after the execution of its last statement
  - since no uncommitted transaction writes are in the DB, NO NEED for **undo**- all values on stable storage are correct (wrt to some point of time)
  - Drawback is that all these uncommitted writes are taking buffer space to the point that other transactions cannot proceed due to the lack of buffer space

395

Failure with Loss of Nonvolatile Storage

- So far we assumed **no loss** of disks
- Technique similar to checkpointing used to deal with loss of disks (non-volatile storage)
- This is called a ***dump***
  - Periodically dump the entire content of the database to stable storage
  - No transaction may be active during the dump procedure (similar to checkpointing)
    - Output all log records currently residing in main memory onto stable storage.
    - Output all buffer blocks onto the disk.
    - Copy the contents of the database to stable storage.
    - Output a record <dump> to log on stable storage.
  - To recover from disk failure
    - restore database from most recent dump.
    - Consult the log and redo all transactions that committed after the dump
- Can be extended to allow transactions to be active during dump; known as fuzzy dump or online dump

EOS

396