Transactions

- A transaction: a sequence of one or more SQL operations (interactive or embedded):
  - declared by the programmer to constitute a unit
  - treated by the DBMS as one unit
Transactions

• A transaction is a sequence of operations on the database and on temporary data, with the following properties:
  – Atomicity: Only successful transactions change the state of the database
  – Isolation: Transactions behave as if they were executed in isolation from each other
  – Durability: If the DBMS crashes after a transaction commits, all effects of the transaction must remain in the database
Transaction management

• Transaction properties:
  – Atomicity, Consistency, Isolation, Durability - ACID
  – The Recovery Manager guarantees Atomicity & Durability.
  – Isolation: Concurrency Control Manager
  – Consistency:
    • Integrity constraints
    • Code correctness
    • Combined with Atomicity, Isolation, Durability

• Isolation and Serializability
Transactions for the DBMS

- For the DBMS, a transaction T makes read/write operations on the database.
- To read a page \( r_i \[x] \) it is first brought into the buffer from the disk, if it is not already in the buffer pool.
- To write a page \( w_i \[x] \) an in-memory copy of the page is first modified and is later written to disk, only when the buffer manager decides to do it.
- This has to be made compatible with Atomicity and Durability.
Lifecycle of a transaction

- Read/Write and Commit are execute by the system when required by the transaction
- The rollback (abort) transition is executed under request or by the system
Kinds of failure

- Transaction failure is an interruption of a transaction which only affects the state of the transaction.
- System failure is a failure of the system (either the DBMS or the computer) which may have affected the content of main memory – persistent store is safe.
- Media failure (aka disaster) also affects persistent store.
Protection from failures

• DB backup.
• Log file (for simplicity assume not buffered):
  – (begin, T)
  – (commit, T)
  – (abort, T)
  – For each write: (write, T, P, BeforeImage, AfterImage)
• LSN
• Recovery: re-execute log operations on the DB backup (details later)
Faster recovery through Checkpoint

• Commit consistent CKP:
  – Do not accept new transactions
  – Wait until all transactions finish
  – Flush all buffer “dirty” pages to disk
  – Write CKP record to the log file

• Buffer consistent CKP - V1:
  – Do not accept new transactions
  – Suspend active transactions
  – Flush all buffer “dirty” pages to disk.
  – Write CKP record (with list of active T ids) to the log file
Faster recovery through Checkpoint

• No stop checkpoint:
  – Write begin-CKP record (with list of active T ids) to the log file
  – Start a new thread that scans the buffer and flush buffer “dirty” pages to disk in parallel with the standard transactions – guaranteeing that all pages that were dirty at begin-CKP time are flushed before end of CKP
  – Write end-CKP record to the log file

• Guarantee:
  – For any end-CKP in the log, every update performed before the corresponding begin-CKP is on disk
ARIES algorithm checkpoint

• Rather then flushing the pages, it stores the information that is needed, at restart time, to know which operations have to be redone, i.e.:
  – for each dirty page:
    • Its address
    • The LSN of the first record that made page dirty
  – For each transaction:
    • Status, and last LSN if active

• Fuzzy checkpoint: empty <begin chkpoint> and, later, the actual checkpoint information record

• ARIES redoes everything and then undoes
Undo, Redo algorithms
Recovery algorithms

• The methods for transactions management differ for the use of the undo and redo algorithms to recover a database after a failure, e.g. how write operations on the DB and commits are managed.
  – Undo–Redo: Steal Policy (a new T may steal the buffer), NoForce Policy (write of buffer is not forced)
  – Undo–NoRedo: Steal, **Force**
  – NoUndo–Redo: **NoSteal (Pin)**, NoForce
  – NoUndo–NoRedo: **NoSteal (Pin)**, **Force**

• Hyp: a write to the log is forced to the permanent memory
Undo - NoUndo

• Constraints on write:
  – NoUndo:
    • Deferred updates
  – Undo:
    • Free update (or immediate update or buffer stealing):
    • Rule for undoing updates: Log Ahead Rule or Write Ahead Log: save the before images before writing
Redo - NoRedo

• Constraint on commit:
  – NoRedo:
    • Deferred commit: all modified pages have to be flushed before commit: force writes
  – Redo:
    • No constraints on commit, immediate commit: commit now, flush when you like (NoForce)
    • Rule for redoing: Save the after images before committing
Shadow Pages: No-undo and No-redo

NoUndo NoRedo: needs the ability to write many pages atomically – shadow pages
Choice among the different solutions

• Undo – Redo is the best one
Example of undo-redo implementation

- **beginTransaction()**
  - T := newTransactionIde();
  - Log.append(begin, T);
  - return(T).

- **write(T, P, V)**
  - Buffer.getAndPinPage(P);
  - BI := page P; AI := V;
  - Log.append(Write, T, P, BI, AI);
  - Buffer.updatePage(P, V);
  - Buffer.unpinPage(P).
Implementation

- **commit**\((T)\)
  - Log.append(commit, T)

- **abort**\((T)\)
  - Log.append(abort, T);
  - for each \((\text{write}, T, P, BI, AI) \in \text{Log} \) with
  - do Buffer.undoPage\((P, BI)\)
Restart

• restart()
  – for each (begin, T) ∈ Log
    do if (commit, T) ∉ Log
      then add(T, listUndo);
    for each (write, T, P, BI, AI) ∈ Log
      order by LSN
      do if T ∈ listUndo then Buffer.undoPage(P, AI)
        else Buffer.redoPage(P, BI)
Recovery (undo-redo)

- Transaction failure:
  - Undo(T)

- System failure
  - Undo/redo

- Media failure
  - Use the DB backup and redo committed transactions
Restart

Start undo  Start redo

t-ckpt  t-fail

t1  t2  t3  t4  t5
Restart with CKP

- **Restart:**
  
  ```plaintext
  ckp=false; toUndo=toRedo={};
  for backward r in log -- rollback
    until (ckp and empty(toUndo)) {
      if r = (commit,T) then toRedo+=={T};
      elsif r = (write,T,x,bi,ai) and not (T in toRedo)
        then {toUndo+=={T}; undo(x,bi)}
      elsif r = (begin,T) then toUndo-={T}
      elsif r = (b-ckp,TList) then {ckp=true;
        toUndo+==TList-toRedo}
    }
  rollForward(toRedo);
  ```
RollForward

• RollForward(toRedo):
  
  – for r in log starting from last begin-ckp until (empty(toRedo)) {
    if r = (commit,T) then toRedo-{T};
    elsif r = (write,T,x,bi,ai) and (T in toRedo)
      then {redo(x,ai)}
  }


Buffer and Log

- Where is restart executed – log or persistent store?
- What happens if there is a failure during restart?
Common optimizations

• Log granularity is at record (or field) level, not at page level
• Log is buffered
• Pages contain the LSN of the last operation executed
• Undo actions are logged
• Each log entry has the LSN of the previous log entry of the same transaction