PURDUE UNIVERSITY | Department of Computer Science

CS 44800: Introduction To Relational Database Systems

Recovery: Checkpointing

Prof. Chris Clifton

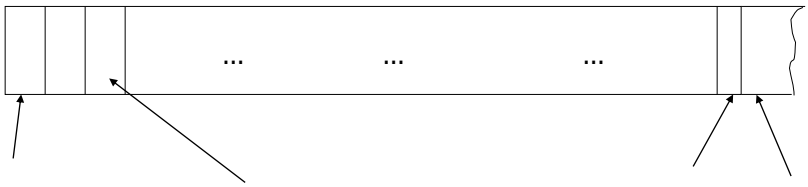
16 November 2021

Indiana Center for Database Systems™

PURDUE UNIVERSITY
Department of Computer Science

Recovery is very, very SLOW !

Redo log:



First Record (1 year ago)

T1 wrote A,B Committed a year ago --> STILL, Need to redo after crash!!

Last Record

PURDUE UNIVERSITY
Department of Computer Science

Solution: Checkpoint (simple version)

- Periodically:

1. Do not accept new transactions
2. Wait until all transactions finish
3. Flush all log records to disk (log)
4. Flush all buffers to disk (DB) (do not discard buffers)
5. Write “checkpoint” record on disk (log)
6. Resume transaction processing

PURDUE UNIVERSITY
Department of Computer Science

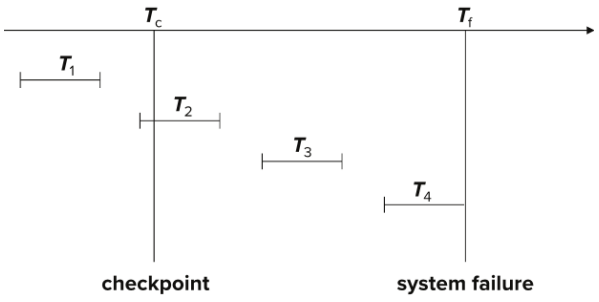
Example: what to do at recovery?

- Redo log (disk):

Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  - Scan backwards from end of log to find the most recent <**checkpoint** *L*> record
  - Only transactions that are in *L* or started after the checkpoint need to be redone or undone
  - Transactions that committed or aborted before the checkpoint already have all their updates output to stable storage.
- Some earlier part of the log may be needed for undo operations
  - Continue scanning backwards till a record <T_i **start**> is found for every transaction T_i in *L*.
  - Parts of log prior to earliest <T_i **start**> record above are not needed for recovery, and can be erased whenever desired.

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

PURDUE UNIVERSITY
Department of Computer Science

Remaining drawbacks:

- *Undo logging:* cannot bring backup DB copies up to date
- *Redo logging:* need to keep all modified blocks in memory until commit

PURDUE UNIVERSITY
Department of Computer Science

Solution: undo/redo logging!

Update $\Rightarrow$ <Ti, Xid, New X val, Old X val>
page X

PURDUE UNIVERSITY
Department of Computer Science

Rules

- Page X can be flushed before or after T_i commit
- Log record flushed before corresponding updated page (WAL)
- Flush at commit (log only)

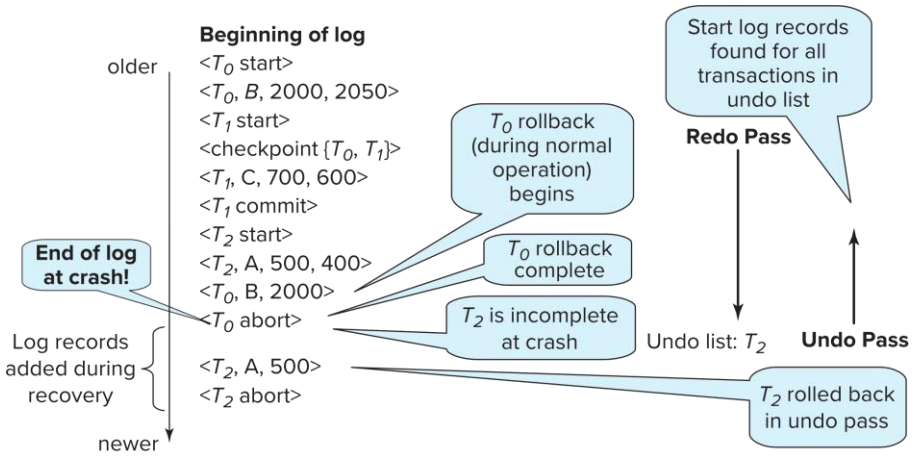
Recovery Algorithm (Cont.)

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete
  - **Undo phase**: undo all incomplete transactions
- **Redo phase**:
  1. Find last <**checkpoint** *L*> record, and set undo-list to *L*.
  2. Scan forward from above <**checkpoint** *L*> record
     1. Whenever a record $<T_i, X_j, V_1, V_2>$ or $<T_i, X_j, V_2>$ is found, redo it by writing V_2 to X_j
     2. Whenever a log record $<T_i$ **start**> is found, add T_i to undo-list
     3. Whenever a log record $<T_i$ **commit**> *or* $<T_i$ **abort**> is found, remove T_i from undo-list

Recovery Algorithm (Cont.)

- **Undo phase:**
  1. Scan log backwards from end
     1. Whenever a log record <T_i, X_j, V_1, V_2> is found where T_i is in undo-list perform same actions as for transaction rollback:
        1. perform undo by writing V_1 to X_j.
        2. write a log record <T_i , X_j, V_1>
     2. Whenever a log record <T_i **start**> is found where T_i is in undo-list,
        1. Write a log record <T_i **abort**>
        2. Remove T_i from undo-list
     3. Stop when undo-list is empty
        1. i.e., <T_i **start**> has been found for every transaction in undo-list
- After undo phase completes, normal transaction processing can commence

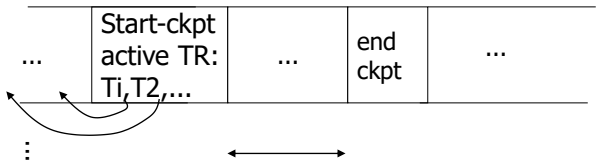
Example of Recovery



PURDUE UNIVERSITY
Department of Computer Science

Non-quiesce checkpoint

L
O
G



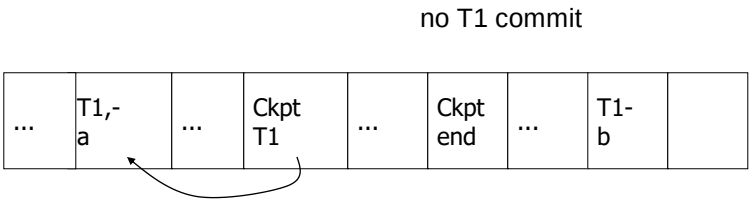
for
undo

dirty buffer
pool pages
flushed

PURDUE UNIVERSITY
Department of Computer Science

Examples: What to do at recovery time?

L
O
G

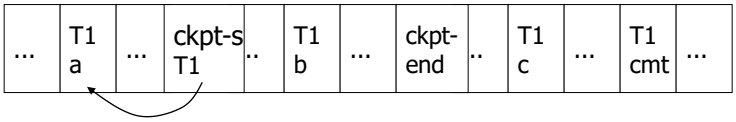


☒ Undo T_1 (undo a,b)

PURDUE UNIVERSITY
Department of Computer Science

Example

LOG

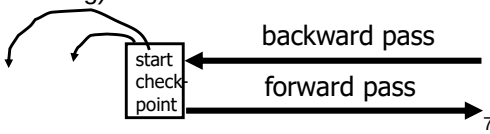


☒ Redo T1: (redo b,c)

PURDUE UNIVERSITY
Department of Computer Science

Recovery process:

- Backwards pass (end of log ➲ latest checkpoint start)
  - construct set S of committed transactions
  - undo actions of transactions not in S
- Undo pending transactions
  - follow undo chains for transactions in (checkpoint active list) - S
- Forward pass (latest checkpoint start ➲ end of log)
  - redo actions of S transactions



Log Record Buffering

- **Log record buffering**: log records are buffered in main memory, instead of being output directly to stable storage.
  - Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Several log records can thus be output using a single output operation, reducing the I/O cost.

Log Record Buffering (Cont.)

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created.
  - Transaction T_i enters the commit state only when the log record <T_i **commit**> has been output to stable storage.
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.
    - This rule is called the **write-ahead logging** or **WAL** rule
    - Strictly speaking, WAL only requires undo information to be output

Database Buffering

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  - If the block chosen for removal has been updated, it must be output to disk
- The recovery algorithm supports the **no-force policy**: i.e., updated blocks need not be written to disk when transaction commits
  - **force policy**: requires updated blocks to be written at commit
    - More expensive commit
- The recovery algorithm supports the **steal policy**: i.e., blocks containing updates of uncommitted transactions can be written to disk, even before the transaction commits

Database Buffering (Cont.)

- If a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first
  - (Write ahead logging)
- No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
  - Before writing a data item, transaction acquires exclusive lock on block containing the data item
  - Lock can be released once the write is completed.
    - Such locks held for short duration are called **latches**.
- **To output a block to disk**
  1. First acquire an exclusive latch on the block
     - Ensures no update can be in progress on the block
  2. Then perform a **log flush**
  3. Then output the block to disk
  4. Finally release the latch on the block

Buffer Management (Cont.)

- Database buffer can be implemented either
  - In an area of real main-memory reserved for the database, or
  - In virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
  - Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot change the partitioning of memory.

Buffer Management (Cont.)

- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - When operating system needs to evict a page that has been modified, the page is written to swap space on disk.
  - When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
    - Known as **dual paging** problem.
  - Ideally when OS needs to evict a page from the buffer, it should pass control to database, which in turn should
    1. Output the page to database instead of to swap space (making sure to output log records first), if it is modified
    2. Release the page from the buffer, for the OS to use
  - Dual paging can thus be avoided, but common operating systems do not support such functionality.

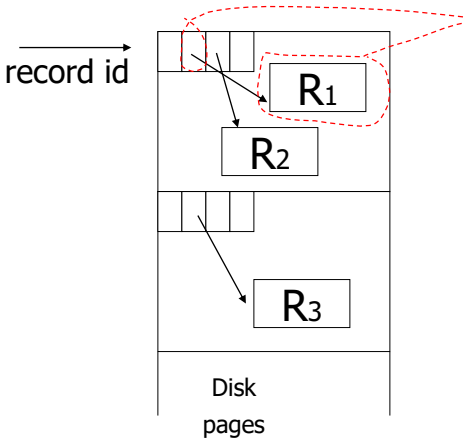
PURDUE UNIVERSITY
Department of Computer Science

Side note: What Needs Locking?

- Multi-granularity – DB/Relation/Block/Record/Field
- But what about other objects?
  - Index?
  - Catalog?
  - Others?
- Index issue: What if we can answer the query without accessing the record?
  - Is there a professor "Clifton"?
  - Index on professor.name

PURDUE UNIVERSITY
Department of Computer Science

Record Locking: Anything Else?



If we lock all data involved in read of R1, we may prevent an update to R2 (which may require reorganization within block)

PURDUE UNIVERSITY
Department of Computer Science

Solution: view DB at two levels

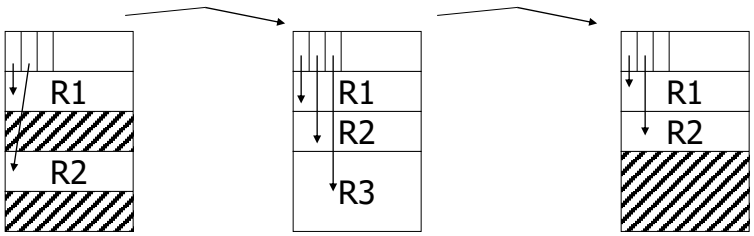
- Top level: record actions
  - lock operations that can be used to determine information about record
    - Record itself
    - Index
- Undo/redo actions are logical operations
  - Insert record(X,Y,Z)
    - Redo: insert(X,Y,Z)
    - Undo: delete
- Low level: deal with physical details
  - latch page during action
  - release at end of action – don't need to follow locking rules

81

PURDUE UNIVERSITY
Department of Computer Science

Undo does not return physical DB to original state; only same logical state

Insert R3 Undo (delete R3)



PURDUE UNIVERSITY
Department of Computer Science

Real world actions

- E.g., dispense cash at ATM
  - Ti = a_1 a_2 …... a_j …... a_n

- Solution
  1. Execute real-world actions after commit
  2. Try to make idempotent

PURDUE UNIVERSITY
Department of Computer Science

Is real-world action idempotent?