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 $<T_i, \text{commit}>$ 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.

---

**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
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).

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

Insert R3  Undo (delete R3)

Real world actions

- E.g., dispense cash at ATM
  - $T_i = a_1 \ a_2 \ \ldots \ a_j \ \ldots \ a_n$

  \[
  \downarrow
  \]
  \[
  \$
  \]

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

ATM

Give\$\$(amt, Tid, time)\$

give(amt)

Failure with Loss of Nonvolatile Storage

- So far we assumed no loss of non-volatile storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
    - 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.
**DB Dump + Log**

- If active database is lost,
  - restore active database from backup
  - bring up-to-date using redo entries in log

---

**Recovering from Failure of Non-Volatile 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**
  - Similar to fuzzy checkpointing
When can log be discarded?

- not needed for media recovery
- not needed for undo after system failure
- not needed for redo after system failure

Remote Backup Systems

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary site is destroyed.
Remote Backup Systems (Cont.)

- **Detection of failure**: Backup site must detect when primary site has failed
  - to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.
  - Heart-beat messages

- **Transfer of control**:
  - To take over control backup site first perform recovery using its copy of the database and all the long records it has received from the primary.
    - Thus, completed transactions are redone and incomplete transactions are rolled back.
  - When the backup site takes over processing it becomes the new primary
  - To transfer control back to old primary when it recovers, old primary must receive redo logs from the old backup and apply all updates locally.

Remote Backup Systems (Cont.)

- **Time to recover**: To reduce delay in takeover, backup site periodically process the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.

- **Hot-Spare** configuration permits very fast takeover:
  - Backup continually processes redo log record as they arrive, applying the updates locally.
  - When failure of the primary is detected the backup rolls back incomplete transactions, and is ready to process new transactions.

- Alternative to remote backup: distributed database with replicated data
  - Remote backup is faster and cheaper, but less tolerant to failure
    - more on this in Chapter 19
What is a Distributed Transaction?

Committing a Distributed Transaction
Remote Backup Systems (Cont.)

- Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.
- **One-safe**: commit as soon as transaction’s commit log record is written at primary
  - Problem: updates may not arrive at backup before it takes over.
- **Two-very-safe**: commit when transaction’s commit log record is written at primary and backup
  - Reduces availability since transactions cannot commit if either site fails.
- **Two-safe**: proceed as in two-very-safe if both primary and backup are active. If only the primary is active, the transaction commits as soon as is commit log record is written at the primary.
  - Better availability than two-very-safe; avoids problem of lost transactions in one-safe.

ARIES: State of the Art

- What we’ve discussed still has a few performance issues
  - Insert/delete can impact multiple records/pages
  - Undo/redo can be expensive (if unnecessary)
- ARIES optimizes this
  1. Uses **log sequence number (LSN)** to identify log records
     - Stores LSNs in pages to identify what updates have already been applied to a database page
  2. Physiological redo
  3. Dirty page table to avoid unnecessary redos during recovery
  4. Fuzzy checkpointing that only records information about dirty pages, and does not require dirty pages to be written out at checkpoint time
     - More coming up on each of the above …
Solution: Add Log Sequence Number

**Log record:**
- LSN=26
- OP=insert(5,v2) into P
- ...

| sem | lsn=25 | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3, v1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3, v1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5, v2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>