Stable storage is an approximation critical for recovery algorithms. It's implemented by **replicating** needed information onto several **nonvolatile storage media** (like disks) with **independent failure modes**. Updates are done in a **controlled manner** to ensure that **failure during data transfer** doesn't damage the essential information.

Shadow Copies: Before updating, create a *complete copy* of the database. Apply updates to the *new copy*, leaving the *original (shadow copy)* untouched. To commit, ensure the new copy is on disk and *atomically update* the `db-pointer` to point to it. To abort, just delete the new copy. It's simple but *expensive* for large databases due to full copying.

Shadow Paging: A variant to *reduce copying*. Uses a *page table* (pointers to data pages). When updating, only the *page table itself* and any *updated pages* are copied. The new page table points to original pages for unchanged data. **Commit involves *atomically updating* the pointer (`db-pointer`)** to the *new page table*. This avoids copying the whole database but *doesn't work well with concurrency* and isn't widely used in databases.

Database Modification:

Updates data in disk buffer/disk (not private memory). Log old/new values *before* modifying.

- **Types:**
    1. **`Immediate`:** Updates while active.
    2. **`Deferred`:** Updates post-commit.
- **Steps:**
    1. Compute privately.
    2. Modify disk buffer block.
    3. Write buffer block to disk.

Q1 Explain why log records for transactions on the undo-list must be processed in reverse order, whereas redo is performed in a forward direction.

Ans :

- **Undo (Reverse Order): To restore the original state *before* a transaction, changes must be undone in the reverse sequence they were applied. Processing forward would incorrectly stop at an intermediate value if an item was updated multiple times.**
- **Redo (Forward Direction): To ensure the final committed state is reached, changes must be reapplied in the same sequence they originally occurred. Processing in reverse would incorrectly leave the data item with an earlier, non-final value.**

Q2. Suppose the deferred modification technique is used in a database.

a. Is the old-value part of an update log record required any more? Why or why not?

b. If old values are not stored in update log records, transaction undo is clearly not feasible. How would the redo-phase of recovery have to be modified as a result?

c. Deferred modification can be implemented by keeping updated data items in local memory of transactions, and reading data items that have not been updated directly from the database buffer. Suggest how to efficiently implement a data item read, ensuring that a transaction sees its own updates.

d. What problem would arise with the above technique, if transactions perform a large number of updates?

Answer: . **Old Value Needed?** No. With **`deferred modification`**, the original value remains untouched in the stable database until commit. Committed transactions don't need undo, and if a transaction aborts or the system crashes before commit, the stable database wasn't altered, making the **`old-value`** in the log redundant.

b. **Redo Phase Modification:** The redo phase simplifies. Since no uncommitted changes have reached the stable database, there's no need to maintain an **`undo-list`** during redo. The redo process only needs to apply log records for committed transactions forward.

c. **Efficient Read Implementation:** To read a data item: First, check the transaction's **`local memory`**. If the item (possibly updated by the transaction) is found, return it. Otherwise, retrieve the item from the **`database buffer`** (loading it into local memory if desired for future reads) and return that value.

d. **Problem with Large Updates:** If a transaction performs many updates, the **`local memory`** required to store all these deferred changes could become excessively large, potentially exhausting available memory or degrading performance.

Q3. Disk space allocated to a file as a result of a transaction should not be released even if the transaction is rolled back. Explain why, and explain how ARIES ensures that such actions are not rolled back.

Ans : **Why not roll back allocation:** Disk space allocated by a transaction shouldn't be released upon rollback because other concurrent transactions might have already stored data in that space. Undoing the allocation would corrupt their data.

How ARIES prevents rollback: ARIES treats such actions (like page allocation) as `nested top actions`. To prevent undo, when the allocation occurs, a special `dummy CLR` (Compensation Log Record) is generated. During rollback, this CLR's `UndoNextLSN` directs the process to `skip` over the original log records for the allocation, effectively making the action permanent even if the main transaction aborts.

Q3. The shadow-paging scheme requires the page table to be copied. Suppose the page table is represented as a B+-tree
. a. Suggest how to share as many nodes as possible between the new copy and the shadow-copy of the B+-tree, assuming that updates are made only to leaf entries, with no insertions and deletions.
b. Even with the above optimization, logging is much cheaper than a shadow-copy scheme, for transactions that perform small updates. Explain why.

Answer a. **Sharing B+-Tree Nodes:** Initially, copy only the **`root node`.** When a **`leaf entry`** is modified, copy only that leaf and the ancestor nodes along the **`path`** back to the (new) root. All unchanged nodes remain **`shared`** between the **`new copy`** and the **`shadow-copy`.**

b. **Why Logging is Cheaper for Small Updates:** Even optimized **`shadow-copy`** schemes require copying multiple pages (the updated leaf and its path to the root) for a single update. **`Logging`** only writes **`small log records`**, which are often grouped onto a few **`log pages`,** resulting in fewer disk I/O operations and potentially less **`disk arm movement`** compared to copying potentially scattered B+-tree pages.

Q4 Disk space allocated to a file as a result of a transaction should not be released even if the transaction is rolled back. Explain why, and explain how ARIES ensures that such actions are not rolled back.

Answer :

Why not roll back allocation: Disk space allocated by a transaction shouldn't be released upon rollback because other concurrent transactions might have already stored data in that space. Undoing the allocation would corrupt their data.

How ARIES prevents rollback: ARIES treats such actions (like page allocation) as `nested top actions`. To prevent undo, when the allocation occurs, a special `dummy CLR` (Compensation Log Record) is generated. During rollback, this CLR's `UndoNextLSN` directs the process to `skip` over the original log records for the allocation, effectively making the action permanent even if the main transaction aborts.