1 Concurrency Control

1. Locking

a. **SKS 18.11** Explain why the following technique for transaction execution may provide better performance than just using strict two-phase locking: First execute the transaction without acquiring any locks and without performing any writes to the database as in the validation-based techniques, but unlike the validation techniques do not perform either validation or writes on the database. Instead, rerun the transaction using strict two-phase locking. (Hint: Consider waits for disk I/O, and how likely it is a deadlock would occur.)

The first phase (execute but don’t do anything) could be viewed as a complete waste of time. But what happens is that it makes sure all the disk blocks needed by the transaction are loaded into memory. If memory is big enough to hold everything needed to run the transaction, and other transactions don’t need memory causing those to be discarded, then when we run the transaction “for real” using strict 2-phase locking, it will run quickly since it never needs to wait for pages to be loaded from disk.

By itself, this doesn’t help - since the “pre-running” would take nearly as long as if we just ran the transaction with strict 2-phase locking to begin with, and we still have to run the real transaction. But if we think about a likely cause of deadlock (or even lock contention without deadlock), we see the advantage. A transaction starts (and acquires some locks). But then it needs to get a page from disk, so another transaction runs (and acquires some locks) while the first one waits on the I/O. In particular, it may find the page requested by the first transaction now in memory, and get locks on that page. But now it runs into a problem getting a lock held by the first transaction, so the database resumes running the first transaction. Which can’t get it’s lock - deadlock. If the data was in memory, the first transaction would likely just run to completion (since it was never waiting for a lock or I/O), so no deadlock - and better overall performance.

b. Google’s Spanner DBMS uses strict two-phase locking for write-transactions, and a timestamp system for read-only transactions. Would we still get serializability of both read-write and read-only transactions if we used non-strict two-phase locking (release locks before commit)? Hint: Think about the advantages of strict two-phase locking, this should put you on the right path.

First, both strict two-phase locking and regular two-phase locking guarantee serializability. So the write transactions are serializable. But what about the read transactions? They have to read “safe” data - the timestamp of the read transaction is between the timestamp of the data item and the “safe time” of the data item. The timestamp of the data is a time after all locks have been acquired, but the safe time is the timestamp of the first completed (but not committed) transaction.

With strict two-phase locking, these rules make sense. But if we release locks before commit, then another transaction could write the data - and perhaps it’s timestamp is very close to the first transaction. If we read only one replica, it could be from either the first or second transaction, and the
timestamp and safe time would both be okay. But this means we could read one replica written by the first transaction, but not yet written by the second - so we have to be serialized between the two. Now imagine we read another data item written by the second transaction, but untouched by the first transaction. This means we need to be serialized after the second transaction. We can’t be both before and after the second transaction.

With strict 2-phase locking, this won’t happen. The timestamp for the second transaction can’t be until after it acquires all it’s locks, which can’t happen until after the first transaction has committed (and its safe time has passed.) So we either can’t read the item written by transaction 2 (it is after our timestamp), or we can’t read the item written by transaction 1 (its safe time is before our timestamp.)

2. Multiple granularity locks

SKS 18.24 The multiple-granularity protocol rules specify that a transaction $T_i$ can lock a node $Q$ in S or IS mode only if $T_i$ currently has the parent of $Q$ locked in either IX or IS mode. Given that SIX and S locks are stronger than IX or IS locks, why does the protocol not allow locking a node in S or IS mode if the parent is locked in either SIX or S mode?

If a parent holds S or SIX, it means there are implicit S locks on all it’s children. So for $Q$, an implicit S Lock already exists and there is no need to lock $Q$ again.

2. Logging and Recovery

1. Logging

SKS 19.21 Consider the log in Figure 19.5 in the book (also slide 19.64 Undo and Redo Logging and Recovery lecture). Suppose there is a crash just before the log record $< T_0 \text{ abort }>$ is written out. Explain what will happen when the system recovers.

Recovery would happen as follows:

Redo phase:

(a) Undo-List = $T_0$, $T_1$

(b) Start from the checkpoint entry and perform the redo operation.

(c) Set $C = 600$

(d) $T_1$ is removed from the Undo-list as there is a commit record.

(e) $T_2$ is added to the Undo list on encountering the $< T_2 \text{ start }>$ record.

(f) Set $A = 400$

(g) Set $B = 2000$

Undo phase:

(a) Undo-List = $T_0$, $T_2$

(b) Scan the log backwards from the end.

(c) Set $A = 500$; write a log record $< T_2, A, 500 >$

(d) Write a log record $< T_2, \text{ abort }>$; remove $T_2$ from undo-list

(e) Set $B = 2000$; write a log record $< T_0, B, 2000 >$

(f) Write a log record $< T_0, \text{ abort }>$; remove $T_0$ from undo-list

(g) Undo-list is empty

At the end of the recovery process, the state of the system is as follows: $A = 500$, $B = 2000$, $C = 600$. The log records added during recovery are:

$< T_2, A, 500 >$

$< T_2, \text{ abort }>$

$< T_0, B, 2000 >$

$< T_0, \text{ abort }>$
2. Logging vs. shadow copy

SKS 19.24 Compare log-based recovery with the shadow-copy scheme in terms of their overheads for the case when data are being added to newly allocated disk pages (in other words, there is no old value to be restored in case the transaction aborts).

The book describes shadow copies as copying the entire database, which is clearly slow - and if you made that assumption, an answer noting that problem is fine. The answer below assumes the more interesting case where we only copy changed blocks, and we have a block mapping table that maps the data in the database to the “current” blocks (referred to as shadow paging in the book).

With shadow-copy based recovery, the newly allocated pages would need to be written to disk before commit (to ensure durability). Commit requires that we write the page mapping table, so that we know that the newly written pages are the “real” ones, rather than the old pages. Log-based recovery needs to record the changes in the log - in this case, just the new values (since there are no old values), and the log records need to be written to disk before commit. Since the log records need to contain the information required to redo the transaction, they presumably take as much space as the actual disk pages - so the number of blocks written would be about the same. We do save one write, since we don’t have to save the page mapping table, so log-based recovery would seem very slightly faster.

However, with log-based recovery, we still have a lot of modified database pages in memory. Eventually these will need to be written to disk, resulting in higher cost AFTER the transaction is committed. With shadow pages, once the transaction is committed, we could discard all of the blocks in memory if we need the space for something else.

If we have small transactions (the entire log for a transaction fits in one block), the two are the same, as writing the log block to disk on commit corresponds to writing the page mapping table to disk on commit. Undo/redo logging commits faster, as we only need to write the log, not the disk blocks and page mapping table. But shadow copies are faster on recovery, since there is nothing to do (any uncommitted transactions, even if their data is written to disks, are not reflected in the pages in the page mapping table.) Logs can come out ahead on total writes as well, if we have a lot of memory - because some blocks may have several transactions update them before they get written to disk (since we can always recover from the log), where shadow paging would require the block be written before each commit.

A difficulty with shadow copies is that we can’t have committed and uncommitted transactions that have written to the same block/page, as the page mapping table would either include uncommitted transactions in the “current” database, or fail to include committed transactions. In practice, we solve this by doing locking at the page level, which lowers concurrency.

3 Views and Triggers

1. Views and triggers

SKS 5.6 Consider the bank database of Figure 5.21. Let us define a view branch_cust as follows:

```sql
create view branch_cust as
    select branch_name, customer_name
    from depositor, account
    where depositor.account number = account.account_number
```

Suppose that the view is materialized; that is, the view is computed and stored. Write triggers to maintain the view, that is, to keep it up-to-date on insertions to depositor or account. It is not necessary to handle deletions or updates. Note that, for simplicity, we have not required the elimination of duplicates.

I’m showing this in (roughly) Oracle syntax, just to give you a feel for different ways triggers may be done in different DBMSs.

```sql
CREATE TRIGGER branch_cust_on_account_insert
AFTER INSERT OR UPDATE OF account_number, branch_name ON account
DECLARE
cust_name VARCHAR;
```
BEGIN
SELECT customer_name INTO cust_name
FROM depositor
WHERE account_number= :new.account_number;

IF (cust_name IS NOT NULL) THEN
    INSERT INTO branch_cust ( branch_name, customer_name ) VALUES (
        :new.branch_name,
        cust_name
    );
END IF;
END branch_cust_on_account_insert;

CREATE TRIGGER branch_cust_on_depositor_insert
AFTER INSERT OR UPDATE OF account_number, customer_name ON depositor
DECLARE
    br_name VARCHAR;
BEGIN
SELECT branch_name INTO br_name
FROM account
WHERE account_number= :new.account_number;

IF (cust_name IS NOT NULL) THEN
    INSERT INTO branch_cust ( branch_name, customer_name ) VALUES (
        br_name,
        :new.customer_name
    );
END IF;
END branch_cust_on_depositor_insert;

The key points are

- Conditions under which the trigger is activated (insert, update into table)
- When it is executed (in this case, after, so that corresponding updates to customer or branch can occur.)
- Getting the needed information
- Updates only to the needed information (e.g., not recreating the entire materialized view).

There are some problems. For starters, a person with multiple accounts will occur multiple times. You might also think that we get multiple entries for a new customer, once when they are inserted into depositor, and once when their account is created in account. But since the trigger occurs after each row, if the depositor is created before the account, select from account will not give a branch_name, and no insert will be done. An alternative semantics (not supported by Oracle) would be to have the triggers executed at the end of the transaction, this would result in duplicates.

Note that we could also use a trigger to allow an insert into the branch_cust ”materialized view”. But the semantics would be challenging. If this is a new customer, it might make sense to create a new account. But if an existing customer, does this mean we move the account from one branch to another, or create an account at the new branch? Most likely, you’d want to set permissions (REVOKE INSERT ON branch_cust) so that users who can modify the data can’t modify branch_cust, and use what Oracle calls an “Invoker’s rights” subprogram. The Invoker (user who created the triggers) would be the only one allowed to update branch_cust.

Any database that supports such complicated triggers should support materialized views, and we wouldn’t have to deal with this. But it is an interesting exercise in triggers.
2. Cascading triggers

SKS 5.20 The execution of a trigger can cause another action to be triggered. Most database systems place a limit on how deep the nesting can be. Explain why they might place such a limit.

While standard SQL queries are guaranteed to terminate (SQL isn't a turing-complete programming language), sets of triggers are not. Imagine that an insert on relation A triggers an insert on B, and an insert on B triggers an insert on A. As can be seen above, triggers can be arbitrarily complex programs in some systems. This means it can be difficult (or even probably impossible - think of the halting problem) to tell if we have cascading triggers that will eventually terminate, or if we have a cycle that will never terminate. One solution is to keep track of the “depth” of trigger invocation (a variable that is increased when a trigger is invoked and decreased when it is done), and if that depth exceeds a threshold, we assume a cycle and abort the transaction. This is kind of like a simple deadlock prevention scheme that may cause an abort even if there isn’t a deadlock - except that with deadlock, it is possible (although possibly expensive) to determine if there is a deadlock, with triggers it may not even be possible to determine if they would terminate.

4 Distributed Databases

1. Two-phase commit

SKS 23.3 Give a type of failure that occurs in a distributed system during 2-Phase Commit (after sending the “ready to commit” message to the leader, but before receiving the final “commit” message) for a transaction. Explain how 2PC ensures transaction atomicity despite the failure.

While you don’t have to answer this, you may find it interesting to think about what happens with Google Spanner with the same failure during the commit (2-PL and 2-phase commit for read/write transactions, timestamp for read-only transactions).

Possible Failures (You only need to mention one of the following failures):

(a) Site failure
(b) Disk Failure
(c) Communication/Link failure, leading to disconnection of one or more sites from the network

A proof that 2PC guarantees atomic commits/aborts in spite of site and link failures follows. The main idea is that after all sites reply with a \( \langle \text{ready T} \rangle \) message, only the coordinator of a transaction can make a commit or abort decision. Any subsequent commit or abort by a site can happen only after it ascertains the coordinator’s decision, either directly from the coordinator or indirectly from some other site. Suppose a site has written the \( \langle \text{ready T} \rangle \) log record, and on inquiry, it found out that some other site has a \( \langle \text{commit T} \rangle / \langle \text{abort T} \rangle \) log record. In this case, it is correct for the site to commit/abort, because that other site would have ascertained the coordinator’s decision (either directly or indirectly) before actually committing/aborting.

2. Write-all approach

SKS 23.15 Give an example where the read one, write all available (as a relaxation of write all) approach leads to an erroneous state.

The problem occurs if the read one happens to read something that wasn’t available when we did our “write all available”. This forces us to serialize before the transaction that did the writing. But we might read something else that it had written, forcing us to serialize after.

Two-phase commit can resolve this, as a replica that wasn’t written by our “write all available” could not be read until it was up to date. This works with a fail-stop model, but if we have a link failure, then the unwritten replica may not even know there was a write to be recovered (since the “write all available” assumed it was dead). Write all guarantees consistency, but means that any failure (link or machine) could prevent a write transaction from completing.