Summary on Concurrency Control techniques (based on chapter 22)

1. What is Concurrency Control?
   - If we insist only one transaction can execute at a time, in serial order, then performance will be quite poor.
   - Concurrency Control is a method for controlling or scheduling the operations of transactions in such a way that concurrent transactions can be executed safely (i.e., without causing the database to reach an inconsistent state).
     - Recall the problems of lost update, Dirty Read, Incorrect Summary
   - If we do concurrency control properly, then we can maximize transaction throughput while avoiding any chance of corrupting the database.
   - Transaction throughput: The number of transactions we can perform in a given time period. Often reported as Transactions per second (TPS) or Transaction per minute (TPM).

2. Locking
   - We need a way to guarantee that the concurrent transactions can be serialized. Locking is one such means.
   - Locks may be applied to data items in two ways:
     - Implicit Locks are applied by the DBMS
     - Explicit Locks are applied by application programs.
   - Locks may be applied to (lock granularity)
     1. a single data item (value)
     2. an entire row of a table
     3. a page (memory segment) (many rows worth)
     4. an entire table
     5. an entire database
   - Many DBMS perform what is called lock escalation where a collection of locks at one level are replaced by fewer locks at larger granularity. For example:
     1. Suppose a table contains 100 rows.
     2. An UPDATE statement begins to work on the table by successively locking rows.
     3. At some point, the lock manager may decide to escalate the locks to the table level. i.e., a single table-level lock is applied and the individual row level locks are released.
   - Lock escalation can reduce the overhead required for transaction processing.
     - Suppose data items a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z are all on the same page.
     - A transaction starts locking a, b, c, d, e, etc. individually. In all, the DBMS would need to take 26 locks.
     - However, if the DBMS recognizes all of these items are on the same page, it might "escalate" the granularity of the locks from the individual item to the page level.
     - Thus only one page level lock would need to be help (in place of the 26 item level locks).
   - There are situations, however, where lock escalation can cause deadlocks to occur.
   - Many DBMS systems support row-level and table-level locking without lock escalation.
   - Locks may be of two types depending on the requirements of the transaction:
     - An Exclusive Lock prevents any other transaction from reading or modifying the locked item.
     - A Shared Lock allows another transaction to read an item but prevents another transaction from writing the item.

3. Two Phased Locking (2PL)
The most commonly implemented locking mechanism is called Two Phased Locking or 2PL. 2PL is a concurrency control mechanism that ensures serializability.

2PL has two phases: Growing and shrinking.
1. A transaction acquires locks on data items it will need to complete the transaction. This is called the growing phase.
2. Once one lock is released, no other lock will be acquired. This is called the shrinking phase. One lock is released at a time.

Example:
User A places an exclusive lock on the balance
User A reads the balance
User A deducts $100 from the balance

User B attempts to place a lock on the balance but fails because A already has an exclusive lock
User B is placed into a wait state
User A writes the new balance of $100
User A releases the exclusive lock on the balance

User B places an exclusive lock on the balance
User B reads the balance
User B deducts $100 from the balance
User B writes the new balance of $100

Example:
User A places a shared lock on item raise_rate
User A reads raise_rate
User A places an exclusive lock on item Amy_salary
User A reads Amy_salary

User B places a shared lock on item raise_rate
User B reads raise_rate

User A calculates a new salary as Amy_salary * (1+raise_rate)

User B places an exclusive lock on item Bill_salary
User B reads Bill_salary
User B calculates a new salary as Bill_salary * (1+raise_rate)
User B writes Bill_salary

User A writes Amy_salary
User A releases exclusive lock on Amy_salary

User B releases exclusive lock on Bill_Salary
User B releases shared lock on raise_rate

User A releases shared lock on raise_rate

Example:
User A places a shared lock on raise_rate
User B attempts to place an exclusive lock on raise_rate
Placed into a wait state
User A places an exclusive lock on item Amy_salary
User A reads raise_rate
User A releases shared lock on `raise_rate`
User B places an exclusive lock on `raise_rate`
User A reads `Amy_salary`
User B reads `raise_rate`
User A calculates a new salary as `Amy_salary * (1+raise_rate)`
User B writes a new `raise_rate`
User B releases exclusive lock on `raise_rate`
User A writes `Amy_salary`
User A releases exclusive lock on `Amy_salary`

- In basic 2PL, transactions get locks as they need them and release them as soon as possible.
- Two variations of 2PL: Conservative and Strict
  - With Conservative 2PL (static 2PL), the transaction pre-declares which items it will work with and it acquires all locks before any work is done. C-2PL is deadlock free
  - With Strict 2PL, once all operations are completed, all of the locks are released after commit. Guarantees strict schedules. S-2PL is not deadlock free. But most popular variation for recovery purposes

**Example:**

Consider T: Ra Wa Rb Wb Rc Wc Rd Wd


Conservative:
TLa TLb TLc Tld TRa Twa TUa TRb TWb TUb TRc TWc TUc TRd TWd TUd

Strict:
TLa TRa Twa TLb TLc TRb TWb TLd TRc TWc TRd TWd TUa TUb TUc TUd

4. Deadlock handling

4.1 Deadlock prevention and avoidance
- Wait-die
  - Based on a nonpreemptive technique and time-stamp ordering
  - If Ti requests a resource currently held by Tj, Ti is allowed to wait only if it has a smaller timestamp than does Tj (Ti is older than Tj)
    Otherwise, Ti is rolled back (dies)
  - Example: Suppose that processes T1, T2, and T3 have timestamps 5, 10, and 15 respectively
    - if T1 requests a resource held by T2, then T1 will wait
    - If T3 requests a resource held by T2, then T3 will be rolled back
- Wound-wait: preemptive

4.2. Deadlock detection and recover

- Based on wait-for graph