Summary on Concurrency Control techniques

Refer to the textbook chapter 22 for additional discussion on this topic.

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 (Transactions per second or Transaction per minute).

2. Locking
   - We need a way to guarantee that the concurrent transactions can be serialized (i.e., to guarantee an acceptable schedule).
   - Locking is commonly used to control access to shared resources (data objects) among concurrent transactions.
   - Locks may be applied to data items in two different ways.
     Implicit Locks are applied by the DBMS
     Explicit Locks are applied by application programs.
   - Locks may be of two types depending on the requirements of the transaction:
     1. An Exclusive Lock prevents any other transaction from reading or modifying the locked item.
     2. A Shared Lock allows another transaction to read an item but prevents another transaction from writing the item.
   - Locking mechanism is implemented by software module called lock manager
     1. Transactions must request lock before performing actions (R or W) on a database object (i.e., data item)
     2. Release lock when done (following locking protocol)
     3. Lock manager cannot always grant requested lock right away (needs to maintain a wait queue)
     4. Maintains a lock table as shown below (as a simplified example)

<table>
<thead>
<tr>
<th>Transaction ID</th>
<th>Record ID</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>T3</td>
<td>R101</td>
<td>Shared (Read)</td>
</tr>
<tr>
<td>T1</td>
<td>R100</td>
<td>Exclusive (Write)</td>
</tr>
<tr>
<td>T2</td>
<td>R101</td>
<td>Shared (Read)</td>
</tr>
</tbody>
</table>

- 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) (multiple rows)
  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 is needed (in stead 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.

3. Two Phased Locking (2PL)

Basic locking protocol
1. If a transaction T wants to use a data item O, it must request a lock on O (Shared, or Exclusive)
2. If another transaction holds lock on O, then T must wait until the lock is released.
3. Lock manager needs to keep the status of the locks on data items, and it also needs to maintain wait queues associated with the locks.
4. Main issue is when the lock should be released.

Simple (naïve) protocol: transaction T releases its lock on O immediately after reading/writing O is done.
1. Does not guarantee serializability (see the example below)
2. Better Solution: 2-Phase Locking

The most commonly implemented locking mechanism is called Two Phase Locking or 2PL Protocol. 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 1:
- T1 acquires an exclusive lock on the balance
- T1 reads the balance
- T1 deducts $100 from the balance

  T2 attempts to acquire a lock on the balance  
    but fails because T1 already holds an exclusive lock
  T2 is placed into a wait state
- T1 writes the new balance of $100
- T1 releases the exclusive lock on the balance

  T2 acquires an exclusive lock on the balance
  T2 reads the balance
  T2 deducts $100 from the balance
  T2 writes the new balance of $100

Example 2:
- T1 acquires a shared lock on item raise_rate
- T1 reads raise_rate
- T1 acquires an exclusive lock on item Bob_salary
- T1 reads Bob_salary

  T2 acquires a shared lock on item raise_rate
  T2 reads raise_rate

  T1 calculates a new salary as Bob_salary * (1+raise_rate)

  T2 acquires an exclusive lock on item John_salary
  T2 reads John_salary
  T2 calculates a new salary as John_salary * (1+raise_rate)
  T2 writes John_salary

  T1 writes Bob_salary
  T1 releases exclusive lock on Bob_salary

  T2 releases exclusive lock on John_Salary
  T2 releases shared lock on raise_rate

  T1 releases shared lock on raise_rate

Example 3:
- T1 acquires a shared lock on raise_rate
- T2 attempts to acquire an exclusive lock on raise_rate  
  Placed into a wait state
- T1 acquires an exclusive lock on item Bob_salary
- T1 reads raise_rate
T1 releases shared lock on raise_rate
T2 acquires an exclusive lock on raise_rate
T1 reads Bob_salary
T2 reads raise_rate
T1 calculates a new salary as Bob_salary * (1+raise_rate)
T2 writes a new raise_rate
T2 releases exclusive lock on raise_rate
T1 writes Bob_salary
T1 releases exclusive lock on Bob_salary

• In basic 2PL, transactions acquire locks as they need them and release them as soon as possible.
• Two variations of 2PL: Conservative and Strict
  o 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 (deadlock prevention based on total ordering, but may experiences poor performance)
  o With Strict 2PL, once all operations are completed, all of the locks are released. In other words, the strict 2 PL holds all locks until the end of transaction. It guarantees serializability but is not deadlock free. It is the most popular variation for recovery purposes (avoid cascading abort)

Example 4:

Consider T: Ra Wa Rb Wb Rc Wc Rd Wd


Conservative:
TLa TLb TLc Tld TRa TWa TUb TRb TWb 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
  o Based on a nonpreemptive technique and time-stamp ordering
  o 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)
  o Example: Suppose that processes T1, T2, and T3 have timestamps 5, 10, and 15 respectively
    ▪ if T1 request 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: lock manager maintains “wait-for” graph, and periodically checks for cycles.
  o If a cycle is detected, abort some transaction to break the cycle (and relinquish its locks)
• Simple approach: If a transaction does not complete within some fixed amount of time. Assume it is deadlocked, and abort.

5. Transactions without locks

• Locking protocol is the most common approach for implementing serializable transactions.
  o Implemented by most real-world systems

• Other approaches
  o Optimistic Concurrency Control:
    ▪ Locking protocols avoid conflicts by waiting (blocking)
    ▪ Optimistic CC protocols let everything run. If conflicts occur, resolve them later by aborting transactions
    ▪ Anticipates (optimistically) that conflicts rarely occur
  o Multiversion Concurrency Control:
    ▪ Make sure transactions never have to wait to read an object
    ▪ Maintain multiple versions of each object, each with a timestamp