Tutorial – 5 (17/Oct/2023)

1. **How does the granularity of data items affect the performance of concurrency control? What factors affect selection of granularity size for data items?**
   The choice of granularity size for data items affects how fine-grained or coarse-grained the locking or isolation of data is.

   a. **Performance Impact:**

   ❖ *Fine-Grained Granularity*: Fine-grained granularity involves locking or isolating individual data items or small sets of data items. This can allow for a higher degree of parallelism, as multiple transactions can operate on different data items concurrently. However, fine-grained locks come with higher overhead due to increased locking and coordination, potentially leading to increased contention and decreased performance.

   ❖ *Coarse-Grained Granularity*: Coarse-grained granularity involves locking or isolating larger portions of data, such as entire tables or chunks of data. While this reduces the overhead of locking and coordination, it limits the level of parallelism, as transactions may need to wait for access to larger chunks of data. However, coarse-grained locks can reduce contention and improve performance in some cases.

   b. **Factors Influencing Granularity Size:**

   ❖ *Access Patterns*: Understanding the typical read and write patterns of data is essential. If specific data items are frequently accessed and modified together, fine-grained locks might be more suitable. If access patterns are more distributed, coarser-grained locks could be preferable.

   ❖ *Contention*: Consider the level of contention among transactions. Fine-grained granularity can lead to more contention, as multiple transactions may compete for access to the same data items. Coarser-grained locks can alleviate contention issues.

   ❖ *System Architecture*: The underlying hardware and software infrastructure can influence the choice of granularity. For example, in a distributed database system, fine-grained locks might introduce excessive network overhead.

   ❖ *Transaction Characteristics*: The nature of the transactions being processed matters. Long-running, complex transactions may benefit from coarser-grained locks to reduce contention. Short and simple transactions might perform better with fine-grained locks.

   ❖ *Data Volume*: The total volume of data in the database can influence granularity. Fine-grained locks can lead to increased overhead when managing a large number of locks, while coarse-grained locks may be more manageable.

   Ultimately, selecting the right granularity size for data items in a database is a balancing act, as there is no one-size-fits-all solution. It depends on the unique characteristics of the data, access patterns, and the goals of the system. Regular performance testing and tuning may be necessary to find the optimal granularity for a specific database.

2. **The MGL (Multiple Granularity Locking) protocol states that a transaction T can unlock a node N, only if none of the children of node N are still locked by transaction T. Show that without this condition, the MGL protocol would be incorrect.**

The MGL (Multiple Granularity Locking) protocol, which allows a transaction T to unlock a node N only if none of the children of node N are still locked by transaction T, is designed to prevent a specific type of anomaly known as "cascading aborts." Cascading aborts occur when the release of a higher-level (parent) lock leads to the need for lower-level (child) locks, and subsequently, the transactions holding those lower-level locks must be aborted. This issue arises in hierarchical locking schemes like MGL when child locks are released before parent locks.

Let's consider an example to demonstrate why the condition that prevents unlocking of a parent node while any child node is still locked is necessary for correctness. Suppose we have a hierarchical data structure, such as a tree, and transactions T1 and T2 as follows:

Transaction T1 locks a parent node P (higher-level lock).
Transaction T2 locks a child node C (lower-level lock) of node P.

Now, imagine that the MGL protocol doesn't include the condition to prevent unlocking the parent node while a child node is still locked. Without this condition, the following sequence of events might occur:

T1 locks node P.
T2 locks node C, a child of node P.
T1 decides to release the lock on node P while T2 still holds the lock on C.
If T1 releases the lock on node P in this scenario, it could lead to a problem:
Since T2 still has the lock on C, it expects to operate on it without interference.
However, node P, which is a parent of C, has been unlocked by T1.

Other transactions can now lock, modify, or delete node P, which is a parent of C. This creates an inconsistency. If T2 continues to operate on C, it may lead to an incorrect result or data corruption because the expected context of C has been altered by T1's unlocking of P.

By enforcing the condition that prevents unlocking the parent node until all child nodes are released, the MGL protocol avoids this issue. It ensures that a parent node's lock is only released when it is guaranteed that no child nodes are still being operated on by other transactions. This helps maintain data integrity and prevents cascading anomalies in hierarchical data structures.

3. **Prove that the basic two-phase locking protocol guarantees conflict serializability of schedules.**
   The basic Two-Phase Locking (2PL) protocol guarantees conflict serializability of schedules by adhering to several key principles:

   - ❖ ***Two-Phase Approach***: 2PL has two phases—growing and shrinking. Transactions can only acquire locks in the growing phase and release locks in the shrinking phase, ensuring no new conflicts are introduced.

   - ❖ ***Lock Point***: Each transaction reaches a lock point when it acquires its final lock, signifying it won't introduce new conflicts.

   - ❖ ***Conflict Resolution***: 2PL uses wait-die or wound-wait schemes to resolve conflicts, maintaining serializability.

   - ❖ ***No Deadlocks***: The protocol avoids cycles in the wait-for graph, preventing deadlocks.

   - ❖ ***Completeness***: If a schedule is conflict-serializable, 2PL ensures it can be generated by following its rules, confirming conflict serializability.

4. **Prove that the wait-die and wound-wait protocols avoid deadlock and starvation.**
   a. Wait-Die Protocol:
   - ❖ *Prevention of Deadlock*: In the wait-die protocol, older transactions are allowed to wait for younger transactions. Younger transactions, on the other hand, are aborted if they request a resource held by an older transaction. This approach ensures that older transactions are not unfairly aborted and can wait for the resource they need.
   - ❖ *Avoiding Starvation*: The wait-die protocol does not guarantee absolute prevention of starvation. Older transactions can wait for younger ones, but it's possible for an older transaction to wait indefinitely if younger transactions continuously request the same resource.

   b. **Wound-Wait Protocol**:
   - ❖ *Prevention of Deadlock*: In this, younger transactions wait for older ones. If a younger transaction requests a resource held by an older transaction, the younger transaction is wounded (aborted). Older transactions are given priority, and younger transactions are terminated if they interfere with older ones.
   - ❖ *Avoiding Starvation*: The wound-wait protocol effectively avoids starvation. Older transactions are not subject to termination, so they can always complete their execution. Younger transactions are wounded and then reattempt their operation. This ensures that no transaction, regardless of age, is indefinitely starved.