Optimistic Methods for Concurrency Control

What is Concurrency Control

Process of managing simultaneous execution of transactions in a shared database, to ensure the serializability of transactions, is known as concurrency control.

Why we need Concurrency Control

Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems:

- Lost Updates.
- Uncommitted Data.
- Inconsistent retrievals.

When we need Concurrency Control

Concurrent access to data is desirable when:

1.The amount of data is sufficiently great that at any given time only fraction of the data can be in primary memory & rest should be swapped from secondary memory as needed.

2. Even if the entire database can be present in primary memory, there may be multiple processes.

Concurrency Control Techniques

- Pessimistic concurrency control
  - Locking
- Optimistic concurrency control

Pessimistic Concurrency Control

- Pessimistic Concurrency Control assumes that conflicts will happen
- Pessimistic Concurrency Control techniques detect conflicts as soon as they occur and resolve them using blocking.

Locking

- Locking is “pessimistic” because it assumes that conflicts will happen.
- The concept of locking data items is one of the main techniques used for controlling the concurrent execution of transactions.
- A lock is a variable associated with a data item in the database. Generally there is a lock for each data item in the database.
- A lock describes the status of the data item with respect to possible operations that can be applied to that item. It is used for synchronising the access by concurrent transactions to the database items.
  - A transaction locks an object before using it
  - When an object is locked by another transaction, the requesting transaction must wait

Disadvantages of locking

•Lock management overhead.

•Deadlock detection/resolution.

•Concurrency is significantly lowered, when congested nodes are locked.

•To allow a transaction to abort itself when mistakes occur, locks can’t be released until the end of transaction, thus currency is significantly lowered

•(**Most Important**) Conflicts are rare. (We might get better performance by not locking, and instead checking for conflicts at commit time.)

Optimistic Concurrency Control

- Optimistic Concurrency Control assumes that conflicts between transactions are rare.
- Does not require locking
- Transaction executed without restrictions
- Check for conflicts just before commit

Optimistic Concurrency Control (Terminology used)

- ReadSet(Ti): Set of objects read by Transaction Ti.
- WriteSet(Ti): Set of objects modified by Transaction Ti.

Phases for Optimistic Concurrency Control

- Read Phase
- Validation Phase
- Write Phase

Read Phase

- No global writes take place
- Whenever the first write to a given object is requested, a copy is made, and all subsequent writes are directed to the copy.
- When the transaction completes, it requests its validation and write phases.

Write Phase

•If the validation fails, the transaction will be backed up and started again as a new transaction

•If validation succeeds, then the transaction enters the write phase where locally written data are made global.

Validation Phase

•Checks are performed to ensure serializibility is not violated if the transaction updates are applied to the database.

•For read only validation consists of checks to ensure that the values read are the current values for the corresponding data items. If not interference has occurred, and the transaction is backed up and restarted.

•For a transaction that has updates, validation consists of determining whether the current transaction leaves the database in a consistent state, with serializibility maintained. If not, transaction is backed up and restarted.

•The scheduling of transactions is done by assigning transaction numbers to transactions

Validation Phase

- Each Transaction Ti is explicitly assigned a unique integer transaction number t(i)
- There must exist a serially equivalend schedule in which transaction Ti comes before transaction Tj whenever $t(i) < t(j)$.

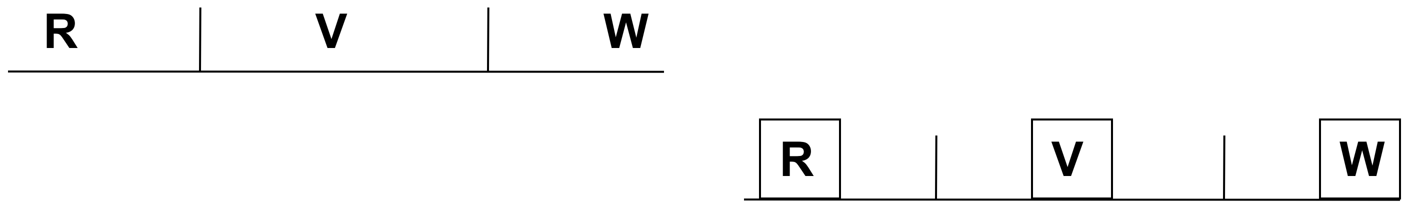
Validation Phase

To guarantee this numbering criteria one of the following three conditions must hold:

1) T_i completes its write phase before T_j starts its read phase.
2) The write set of T_i does not intersect the read set of T_j, and T_i complete its write phase before T_j starts its write phase.
3) The write set of T_i does not intersect the read set or write set of T_j, and T_i completes its read phase before T_j complete its read phase.

Condition 1

Condition 1 states that Ti actually completes before Tj starts, i.e they execute completely in serial order



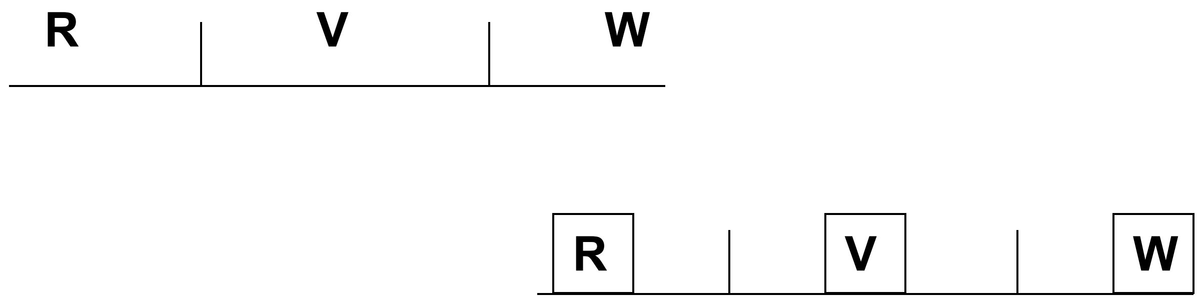
Condition 2

Condition 2 states that the writes of Ti do not affect the read phase of Tj, and that Ti finishes writing before Tj starts writing.

WS(T_i) disjoint from RS(T_j)

Ti does not overwrite Tj

Tj does not read any dirty data since Tj read data when Ti is still modifying it.



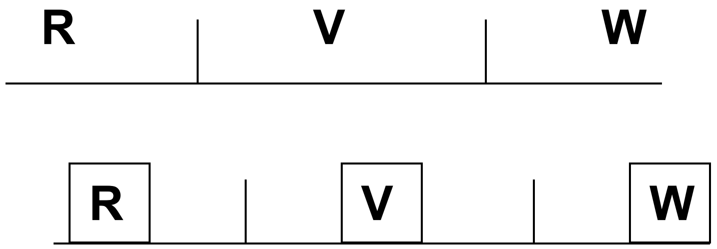
Condition 3

Condition 3 is similar to Condition 2 but does not require that Ti finish writing before Tj starts writing; it simply requires that Ti does not affect the read phase or the write phase of Tj

–WS(T_i) disjoint from RS(T_j)

This condition allows Ti and Tj to write objects at the same time, but there is no overlapping for the sets written by these two transactions.

–WS(T_i) disjoint from WS(T_j).



Assigning Transaction Numbers

•Transactions numbers should be assigned in order, since if Ti completes before Tj starts Ti < Tj.

Solution: Maintain a global counter, when transaction number is needed increment the counter and return the value.

•Transaction number must be assigned somewhere before validation, since validation require knowledge of the transaction number for the transaction being validated.

– Transaction numbers can be assigned at the beginning of the read phase. But this is not optimistic.

– Therefore these are usually assigned at the end of read phase

Practical Considerations

•Space for WriteSets: To validate, Tj must have WriteSets for all Ti where Ti < Tj and Ti was active when Tj began. There may be many such transactions, and we may run out of space.

Solution:

– Concurrency Control maintain some finite number of most recent write sets, where the number is large enough to validate almost all transactions

– If old write sets are unavailable the validation fails and the transaction is backed up.

Practical Considerations

•Starvation: What should be done when validation repeatedly fails ?

Solution:

– If the concurrency control detects a starving transaction, it will be restarted, but without releasing the critical section semaphore, and transaction is run to the completion by write locking the database

Serial Validation

- Implementation of Condition number 1 and 2. Since Condition 3 is not used, last part of condition 3 implies that write phases must be serial (writes won't be interleaved).
  - Ti finishes writing before Tj starts reading (serial)
  - WS(Ti) disjoint from RS(Tj), and Ti finish writing before Tj starts writing.

Serial Validation (implementation)

- Place the assignment of a transaction number, validation and the subsequent write phase all in a critical section. (Since nothing else goes on concurrently, no need to check for Condition 3)
  - Write sets of all transactions started form the beginning to the end of this transaction are checked to see whether they intersect with the read set of current transaction.
  - If validation succeed write phase is committed, otherwise transaction is backed up.
- Optimization for Read-only Transactions:
  - Since there is no write phase we don’t need critical section

Parallel Validation

•Parallel validation uses all 3 validation Conditions, thus allowing greater concurrency (For allowing interleaved writes).

– Maintain a set of transactions ids active for transactions that have completed their read phase but have not yet completed their write phase.

– During validation, in addition to the rules for serial validation, active set's write set is checked for intersections with the read & write sets of the current transaction.

– If validation succeed write phase is committed, otherwise transaction is backed up.

Conclusion

- Optimistic Concurrency Control is superior to locking methods for systems where transaction conflict is highly unlikely, e.g query dominant systems.
  - Avoids locking overhead
  - Using parallel validation OCC can take full advantage of multiprocessor environment.