Concurrency Control

Prof. Weining Zhang

Introduction

- Concurrency control subsystem schedules database operations of concurrent transactions to avoid interference among transactions
- We consider conflict serializable schedules
- Given a schedule, we need to know if it is conflict serializable
- We also need a way to create conflict serializable schedule of transactions without seeing complete transactions
  - Two phase locking protocol

Conflict Operations

- Two operations are *conflict* if they
  - belong to different transactions
  - access same data item
  - involve a write operation
- Types of conflict operations:
  - W-R conflict (dirty read): W1(X), R2(X)
  - R-W conflict (unrepeatable read): R1(X), W2(X)
  - W-W conflict (lost update): W1(X), W2(X)

Ordering Conflict Operations

- Two schedules starting at the same initial state can end up with different final states if they order a pair of conflict operations differently

T1: R(X), X=X*2, W(X), R(Y), Y=Y-5, W(Y)

T2: R(X), X=X+10, W(X)

X=20, Y=35 X=50, Y=30

S1: $R_1(X)\ W_1(X)\ R_2(X)\ W_2(X)\ R_1(Y)\ W_1(Y)$

X=30, Y=30

S3: $R_1(X)\ R_2(X)\ W_1(X)\ R_1(Y)\ W_1(Y)\ W_2(X)$

Conflict Equivalent Schedules

- Two schedules are *conflict equivalent* if they
  - Contain same set of operations (of same set of transactions)
  - Every pair of conflict operations has the same order

T1: R1(X), W1(X), R1(Y), W1(Y)

T2: R2(X), W2(X)

S1: $R_1(X)\ W_1(X)\ R_2(X)\ W_2(X)\ R_1(Y)\ W_1(Y)$

S2: $R_1(X)\ W_1(X)\ R_1(Y)\ W_1(Y)\ R_2(X)\ W_2(X)$

Conflict Serializability

- A schedule S of N transactions is conflict serializable if it is equivalent to a serial schedule of the N transactions.
- S1 above is conflict serializable.
- Serializability can be tested:
  - Create a *precedence graph* of a schedule
  - If the graph has a circle, the schedule is not serializable. Otherwise, it is.
  - Equivalent serial schedule can be obtained by a *topological sort*

Lock-based CC Protocols

- It is not practical to test for serializability during transaction execution
- One way for DBMS to generate serializable schedule is to maintain locks on data items.
- Transactions must request a lock on a data item before accessing the data, and release the lock after using the data
- DBMS grants locks based on a locking protocol

Types of Locks

- Different types of DB operations need different types of locks
  - Read (shared) lock: for read only access
  - Write (exclusive) lock: for read and write
- Notation used in schedules
  - $RL_i(X)$: Ti requests read lock on X
  - $WL_i(X)$: Ti requests write lock on X
  - $UL_i(X)$: Ti release lock on X

Lock Compatibility

- Different transactions may request different types of locks on same data item
- DBMS uses a lock compatibility table to determine if a lock request can be granted.

Lock Requested (columns) / **Lock Held** (rows)

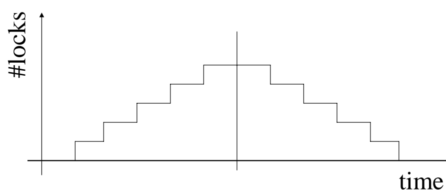
	RL	WL
RL	yes	no
WL	no	no

Two-Phase Locking (2PL) Protocol

- A data can be accessed only if appropriate locks are requested and granted
  - Shared lock for reads
  - Exclusive lock for writes
- A lock request can be granted only if it is compatible with locks held on the data
- *A transaction can not request any more locks after it releases any lock*
- Locks can be released at any time

Two-Phases In 2PL

- *Growing-phase: request locks*
- *Shrinking-phase: release locks*



Example: 2PL

- Not 2PL:
  T1: rl(X), R(X), ul(X), wl(Y), R(Y), W(Y), ul(Y)
- 2PL:
  T2: rl(X), R(X), wl(Y), R(Y), ul(X), W(Y), ul(Y)

Example: 2PL

T1: R(X), X=X+1, W(X), R(Y), Y=Y-1, W(Y)
T2: R(X), X=2*X, W(X), R(Y), Y=2*Y, W(Y)

S: $WL_1(X)R_1(X)W_1(X)WL_1(Y)UL_1(X)WL_2(X)$
$R_2(X)W_2(X)\ R_1(Y)W_1(Y)UL_1(Y)WL_2(Y)$
$UL_2(X)R_2(Y)W_2(Y)\ UL_2(Y)$

- S satisfies the 2PL protocol.
- Theorem: Any schedule produced by 2PL protocol is conflict serializable. (why?)

Problems of Basic 2PL

- Basic 2PL may cause cascading abort

S: $WL_1(X)R_1(X)W_1(X)WL_1(Y)UL_1(X)WL_2(X)$
$R_2(X)W_2(X)\ R_1(Y)\ ABORT_1\ W_1(Y)UL_1(Y)WL_2(Y)$
$UL_2(X)R_2(Y)W_2(Y)\ UL_2(Y)$

T2 must be aborted, too.

- Basic 2PL may cause deadlock.
  - A *deadlock* is a situation in which a set of transactions wait for each other to release locks & no one can make any progress

Example: Deadlock

T1: R(X), X=X+1, W(X), R(Y), Y=Y-1, W(Y)
T2: R(Y), Y=2*Y, W(Y), R(X), X=2*X, W(X)

S': $WL_1(X)R_1(X)WL_2(Y)R_2(Y)W_1(X)W_2(Y)WL_1(Y)$
$WL_2(X)$ Deadlock $R_1(Y)R_2(X)W_2(X)UL_2(Y)$
$W_1(Y)UL_1(X)UL_1(Y)UL_2(X)$

- None of T1 & T2 can make any progress

Deadlock Management

- How can DBMS tell if there is deadlock?

S': $WL_1(X)R_1(X)WL_2(Y)R_2(Y)W_1(X)W_2(Y)WL_1(Y)$
$WL_2(X)$ Deadlock $R_1(Y)R_2(X)W_2(X)UL_2(Y)$
$W_1(Y)UL_1(X)UL_1(Y)UL_2(X)$

Wait-for-graph has a cycle!

Deadlock Management Options

- Deadlock prevention
  - Conservative 2PL
  - Resource Ordering
  - Transaction prioritization
- Deadlock detection and resolution
  - Deadlock detection
  - Deadlock resolution

Transaction prioritization

- Assign a priority to each transaction and avoid potential deadlocks by aborting transactions with lower priorities.
- One priority implementation is to use the start time (*time stamp*) of each transaction, i.e., older transactions have higher priority.
- Two possible rules for choosing a transaction to abort: wait-die & wound-wait.

Wait-Die Rule

- When Ti requests an incompatible lock held by Tj
  - if Ti is *older* than Tj, then Ti *waits*;
  - if Ti is *younger* than Tj, then Ti *dies* (i.e.,Ti is aborted and restarted with its old time stamp)
- Wait-Die Rule prevents both deadlock and starvation (why?)
  - *Starvation* : a transaction is repeatedly aborted in favor of other transactions

Wound-Wait Rule

- When Ti requests an incompatible lock held by Tj,
  - if Ti is *older* than Tj, then Ti *wounds* Tj (i.e., aborting Tj & gets the lock; Tj will restart with the same time stamp.)
  - if Ti is *younger* than Tj, then Ti *waits*;
- Does Wound-Wait Rule prevent deadlock and starvation?

Example: Transaction prioritization

- Assume T1 is older then T2

T1: R(X) W(X) R(Y) W(Y)
T2: R(Y) W(Y) R(X) W(X)

S: $WL_1(X)$ $R_1(X)$ $W_1(X)$ $WL_2(Y)$ $R_2(Y)$ $W_2(Y)$
$WL_1(Y)$ $R_1(Y)$ $W_1(Y)$ $UL_1(X)$ $UL_1(Y)$ $WL_2(X)$
Deadlock $R_2(X)$ $W_2(X)$ $UL_2(Y)$ $UL_2(X)$

- Wait-Die: T1 waits for T2 on Y, T2 dies on X
- Wound-Wait: T2 is aborted, T1 gets lock on Y
- Either way, S is avoided

Summary of Concurrency Control

- 2PL is a simple and practical mechanism to get serializability
  - Strict 2PL is also recoverable, and has no cascade abort
- Lock manager module automates 2PL so that only the access methods worry about it.
  - Lock table is a big main-memory hash table
- Deadlocks are possible, and can be either avoid or resolved
- Other CC mechanisms: timestamp, validation, …

Look Ahead

- Next topic: Crash Recovery
- Read textbook:
  - Chapter 19