

**Transaction Processing:
Concurrency Control**

CPS 216
Advanced Database Systems

ACID

- Atomicity
 - Transactions are either done or not done
 - They are never left partially executed
- Consistency
 - Transactions should leave the database in a consistent state
- Isolation
 - Transactions must behave as if they were executed in isolation
- Durability
 - Effects of completed transactions are resilient against failures

2

Transaction in SQL

- (Implicit beginning of transaction)
SELECT ...;
UPDATE ...;
.....
ROLLBACK | COMMIT;
- ROLLBACK (a.k.a. transaction abort)
 - Will undo the the partial effects of the transaction
 - May be initiated by the DBMS

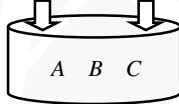
3

Concurrency control

- Goal: ensure the “I” (isolation) in ACID

T_1 :
 read(A);
 write(A);
 read(B);
 write(B);
 commit;

T_2 :
 read(A);
 write(A);
 read(C);
 write(C);
 commit;



4

Good versus bad schedules

| T_1 | T_2 | T_1 | T_2 | T_1 | T_2 |
|-------|-------|-------|-------|-------|-------|
| r(A) | | r(A) | | r(A) | |
| w(A) | | | r(A) | w(A) | |
| r(B) | | w(A) | | | r(A) |
| w(B) | | | w(A) | | w(A) |
| | r(A) | r(B) | | r(B) | |
| | w(A) | | r(C) | | r(C) |
| | r(C) | w(B) | | w(B) | |
| | w(C) | | w(C) | | w(C) |

5

Serial schedule

- Execute transactions in order, with no interleaving of operations
 - $T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B), T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C)$
 - $T_2.r(A), T_2.w(A), T_2.r(C), T_2.w(C), T_1.r(A), T_1.w(A), T_1.r(B), T_1.w(B)$
 - Isolation achieved by definition!
- Problem: no concurrency at all
- Question: how to reorder schedule to allow more concurrency

6

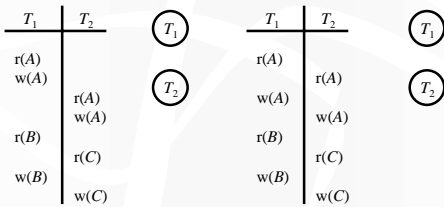
Conflicting operations

- Two operations on the same data item conflict if at least one of the operations is a write
 - $r(X)$ and $w(X)$ conflict
 - $w(X)$ and $r(X)$ conflict
 - $w(X)$ and $w(X)$ conflict
 - $r(X)$ and $r(X)$ do not
 - $r/w(X)$ and $r/w(Y)$ do not
- Order of conflicting operations matters
 - If $T_1.r(A)$ precedes $T_2.w(A)$, then conceptually, T_1 should precede T_2

7

Precedence graph

- A node for each transaction
- A directed edge from T_i to T_j if an operation of T_i precedes and conflicts with an operation of T_j in the schedule



8

Conflict-serializable schedule

- A schedule is conflict-serializable iff its precedence graph has no cycles
- A conflict-serializable schedule is equivalent to some serial schedule (and therefore is “good”)
 - In that serial schedule, transactions are executed in the topological order of the precedence graph
 - You can get to that serial schedule by repeatedly swapping adjacent, non-conflicting operations from different transactions

9

Locking

- Rules

- If a transaction wants to read an object, it must first request a shared lock (S mode) on that object
- If a transaction wants to modify an object, it must first request an exclusive lock (X mode) on that object
- Allow one exclusive lock, or multiple shared locks

Mode of the lock requested

| | | | | |
|--|---|---|---|-----------------|
| Mode of lock(s) currently held by other transactions | S | S | X | Grant the lock? |
| S | S | X | | |
| X | S | X | | |

Compatibility matrix

10

Basic locking is not enough

| | T_1 | T_2 |
|--------|-------|-------|
| $r(A)$ | | |
| $w(A)$ | | |
| $r(A)$ | | |
| $w(A)$ | | |
| $r(B)$ | | |
| $w(B)$ | | |
| $r(B)$ | | |
| $w(B)$ | | |

11

Two-phase locking (2PL)

- All lock requests precede all unlock requests

- Phase 1: obtain locks, phase 2: release locks

| T_1 | T_2 | | T_1 | T_2 |
|--------|--------|--|--------|--------|
| $r(A)$ | | 2PL guarantees a conflict-serializable schedule → | $r(A)$ | |
| $w(A)$ | | | $w(A)$ | |
| | $r(A)$ | | $r(A)$ | $w(A)$ |
| | $w(A)$ | | $r(B)$ | $w(B)$ |
| | $r(B)$ | | $r(B)$ | $w(B)$ |
| | $w(B)$ | | | |

12

Problem of 2PL

| T_1 | T_2 |
|--------|-------|
| r(A) | |
| w(A) | r(A) |
| | w(A) |
| r(B) | |
| w(B) | r(B) |
| | w(B) |
| Abort! | |

- T_2 has read uncommitted data written by T_1
- If T_1 aborts, then T_2 must abort as well
- Cascading aborts possible if other transactions have read data written by T_2

- What's worse, what if T_2 commits before T_1 ?

13

Strict 2PL

- Only release locks at commit/abort time
 - A writer will block all other readers until the writer commits or aborts
- Used in most commercial DBMS (except Oracle)

14

Deadlocks

| T_1 | T_2 |
|-------|-------|
| r(A) | |
| w(A) | r(B) |
| | w(B) |
| r(B) | r(A) |
| w(B) | w(A) |

Deadlock!

Deadlock = cycle in the wait-for graph



15

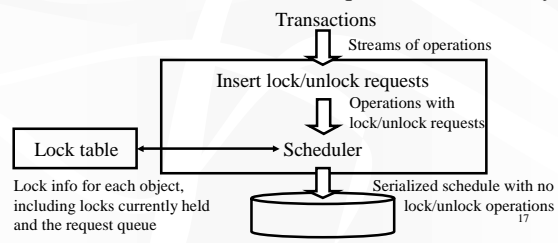
Dealing with deadlocks

- Impose an order for locking objects
 - Must know in advance which objects a transaction will access
- Timeout
 - If a transaction has been blocked for too long, just abort
- Prevention
 - Idea: abort more often, so blocking is less likely
 - Wait/die versus wound/wait
- Detection using wait-for graph
 - Idea: deadlock is rare, so only deal it when it becomes an issue
 - How often do we detect deadlocks?
 - Which transactions do we abort in case of deadlock?

16

Implementation of locking

- Do not rely on transactions themselves to lock/unlock explicitly
- DBMS inserts lock/unlock requests automatically



SQL transaction isolation levels

- SERIALIZABLE (default)
- Weaker isolations levels
 - READ UNCOMMITTED
 - READ COMMITTED
 - REPEATABLE READ
- Why weaker levels?

18

READ UNCOMMITTED

- Dirty reads possible (dirty = uncommitted)

- Example: wrong average

| | |
|-----------------------------|---------------------|
| T1: | T2: |
| UPDATE Account | |
| SET balance = balance - 200 | |
| WHERE number = 142857; | SELECT AVG(balance) |
| | FROM Account; |
| ROLLBACK; | COMMIT; |

- Possible cause

19

READ COMMITTED

- No dirty reads, but non-repeatable reads possible

- Example: different averages

| | |
|-----------------------------|---------------------|
| T1: | T2: |
| UPDATE Account | SELECT AVG(balance) |
| SET balance = balance - 200 | FROM Account; |
| WHERE number = 142857; | |
| COMMIT; | SELECT AVG(balance) |
| | FROM Account; |
| | COMMIT; |

- Possible cause

20

REPEATABLE READ

- Reads repeatable, but may see phantoms

- Example: different average (still!)

| | |
|-----------------------|---------------------|
| T1: | T2: |
| INSERT INTO Account | SELECT AVG(balance) |
| VALUES(428571, 1000); | FROM Account; |
| COMMIT; | |
| | SELECT AVG(balance) |
| | FROM Account; |
| | COMMIT; |

- Possible cause

21

Summary of SQL isolation levels

| Isolation level / anomaly | Dirty reads | Non-repeatable reads | Phantoms |
|---------------------------|-------------|----------------------|----------|
| READ UNCOMMITTED | Yes | Yes | Yes |
| READ COMMITTED | No | Yes | Yes |
| REPEATABLE READ | No | No | Yes |
| SERIALIZABLE | No | No | No |

- Criticized for definition in terms of anomalies
 - Berenson, Bernstein, Gray, et al. “A critique of ANSI SQL isolation levels,” SIGMOD 1995

22

Concurrency control without locking

- Optimistic (validation-based)
- Timestamp-based
- Multi-version (Oracle)

23

Optimistic concurrency control

- Locking is pessimistic
 - Use blocking to avoid conflicts
 - Overhead of locking even if contention is low
- Optimistic concurrency control
 - Assume that most transactions do not conflict
 - Let them execute as much as possible
 - If it turns out that they conflict, abort and restart

24

Sketch of protocol

- Read phase: transaction executes, reads from the database, and writes to a private space
- Validate phase: DBMS checks for conflicts with other transactions; if conflict is possible, abort and restart
 - Requires maintaining a list of objects read and written by each transaction
- Write phase: copy changes in the private space to the database

25

Pessimistic versus optimistic

- Overhead of locking vs. overhead of validation and copying private space
- Blocking versus aborts and restarts
- Agrawal, Carey, and Livny. “Concurrency control performance modeling: alternatives and implications,” TODS 12(4), 1987
 - Locking has better throughput for environments with medium-to-high contention
 - Optimistic concurrency control is better when resource utilization is low enough

26

Timestamp-based

- Associate each database object with a read timestamp and a write timestamp
- Assign a timestamp to each transaction
 - Timestamp order is commit order
- When transaction reads/writes an object, check the object’s timestamp for conflict with a younger transaction; if so, abort and restart
- Problems
 -
 -

27

Multi-version concurrency control

- Maintain versions for each database object
 - Each write creates a new version
 - Each read is directed to an appropriate version
 - Conflicts are detected in a similar manner as timestamp concurrency control
- In addition to the problems inherited from timestamp concurrency control
 - Pro:
 - Con:
- Oracle uses some variant of this scheme

28

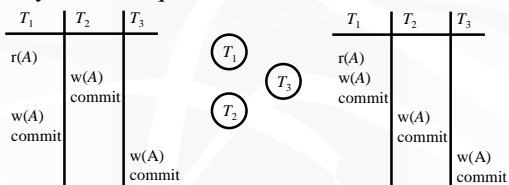
Summary

- Covered
 - Conflict-serializability
 - 2PL, strict 2PL
 - Deadlocks
 - Overview of other concurrency-control methods
- Not covered
 - View-serializability (overview next)
 - Hierarchical locking (overview next)
 - Predicate locking and tree locking (later in course)

29

View-serializability

- Some schedules are not conflict-serializable, but they are still equivalent to a serial schedule



- View-serializability allows blind writes
 - That's it? Forget it!

30

Hierarchical locking

- A database contains many tables, a table contains many pages/blocks, and a page/block contains many rows...
- Fine-granule locking allows more concurrency
 - Example:
- Coarse-granule locking has lower overhead
 - Example:
- Allow both: need to revise the locking protocol

31

Next time

Recovery

SQL triggers and programming interface

32
