Database Systems
(資料庫系統)

December 16, 2009

1

Announcement

- The correct (compile-able) version of Assignment #5 (B+ Tree) was on the course homepage last Friday.
  - It is due next Wed

2

Overview of Transaction Management

Chapter 16

3

Transaction

- Transaction = one execution of a user program.
  - Example: transfer money from account A to account B.
  - A Sequence of Read & Write Operations
- How to improve system throughput (# transactions executed per time unit)?
  - Make each transaction execute faster (faster CPU? faster disks?)
  - What's the other option?

4

Concurrent Execution

- Why concurrent execution is needed?
  - Disk accesses are slow and frequent - keep the CPU busy by running several transactions at the same time.
- Have you written multi-threaded programs? What is so difficult about writing them?
  - Concurrency Control ensures that the result of concurrent execution of several transactions is the same as some serial (one at a time) execution of the same set of transactions.
  - How can the results be different?

5

Non-serializable Schedule

- A is the number of available copies of a book = 1.
- T1 wants to buy one copy.
- T2 wants to buy one copy.
- What is the problem here?
  - T1 gets an error.
    - The result is different from any serial schedule T1,T2 or T2,T1
- How to prevent this problem?
  - How to detect a schedule is serializable?

T1	T2
R(A=1)	
Check if (A>0)	
	R(A=1)
	Check if (A>0)
	W(A=0)
W(A=0)	
	Commit
Commit	

6

System crash

- Must also handle system crash in the middle of a transaction (or aborted transactions).
  - Crash recovery ensures that partially aborted transactions are not seen by other transactions.
  - How can the results be seen by other transactions?

7

Unrecoverable Schedule

- T1 deducts $100 from A.
- T2 adds 6% interests to A.
- T2 commits.
- T1 aborts.
- Why is the problem?
  - Undo T1 => T2 has read a value for A that should never been there.
  - But T2 has committed! (may not be able to undo committed actions).
  - This is called unrecoverable schedule.
- Is this a serializable schedule?
- How to ensure/detect that a schedule is recoverable (& serializable)?

T1	T2
R(A)	
W(A-100)	
	R(A)
	W(A+6%)
	Commit
Abort	

8

Outline

- Four fundamental properties of transactions (ACID)
  - Atomic, Consistency, Isolation, Durability
- Schedule actions in a set of concurrent transactions
- Problems in concurrent execution of transactions
- Lock-based concurrency control (Strict 2PL)
- Performance issues with lock-based concurrency control

9

ACID Properties

- The DBMS's abstract view of a transaction: a sequence of read and write actions.
- Atomic: either all actions in a transaction are carried out or none.
  - Transfer $100 from account A to account B: R(A), A=A-100, W(A), R(B), B=B+100, W(B) => all actions or none (retry).
  - The system ensures this property.
  - How can a transaction be incomplete?
    - Aborted by DBMS, system crash, or error in user program.
  - How to ensure atomicity during a crash?
    - Maintain a log of write actions in partial transactions. Read the log and undo these write actions.

10

ACID Properties (2)

- Consistency: run by itself with no concurrent execution leave the DB in a "good" state.
  - This is the responsibility of the user.
  - No increase in total balance during a transfer => the user makes sure that credit and debit the same amount.

11

ACID Properties (3)

- Isolation: transactions are protected from effects of concurrently scheduling other transactions.
  - The system (concurrency control) ensures this property.
  - The result of concurrent execution is the same as some order of serial execution (no interleaving).
  - T1 || T2 produces the same result as either
    - T1;T2 or T2;T1.

12

ACID Properties (4)

- Durability: the effect of a completed transaction should persist across system crashes.
  - The system (crash recovery) ensures durability property and atomicity property.
  - What can a DBMS do to ensure durability?
  - Maintain a log of write actions in partial transactions. If the system crashes before the changes are made to disk from memory, read the log to remember and restore changes when the system restarts.

13

Schedules

- A transaction is seen by DBMS as a list of read and write actions on DB objects (tuples or tables).
  - Denote $R_T(O)$, $W_T(O)$ as read and write actions of transaction T on object O.
- A transaction also needs to specify a final action:
  - commit action means the transaction completes successfully.
  - abort action means to terminate and undo all actions
- A schedule is an execution sequence of actions (read, write, commit, abort) from a set of transactions.
  - A schedule can interleave actions from different transactions.
  - A serial schedule has no interleaving actions from different transactions.

14

Examples of Schedules

Schedule with Interleaving Execution

T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
	Commit
R(C)	
W(C)	
Commit	

Serial Schedule

T1	T2
R(A)	
W(A)	
R(C)	
W(C)	
Commit	
	R(B)
	W(B)
	Commit

15

Concurrent Execution of Transactions

- Why do concurrent executions of transactions?
  - Better performance.
  - Disk I/O is slow. While waiting for disk I/O on one transaction (T1), switch to another transaction (T2) to keep the CPU busy.
- System throughput: the average number of transactions completed in a given time (per second).
- Response Time: difference between transaction completion time and submission time.
  - Concurrent execution helps response time of small transaction (T2).

T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
	Commit
R(C)	
W(C)	
Commit	

16

Serializable Schedule

- A serializable schedule is a schedule that produces identical result as some serial schedule.
  - A serial schedule has no interleaving actions from multiple transactions.
- We have a serializable schedule of T1 & T2.
  - Assume that T2:W(A) does not influence T1:W(B).
  - It produces the same result as executing T1 then T2 (denote T1;T2).

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	Commit
Commit	

17

Anomalies in Interleaved Execution

- Under what situations can an arbitrary (non-serializable) schedule produce inconsistent results from a serial schedule?
  - Three possible situations in interleaved execution.
    - Write-read (WR) conflict
    - Read-write (RW) conflict
    - Write-write (WW) conflict
  - Note that you can still have serializable schedules with the above conflicts.

18

WR Conflict (Dirty Read)

- Situation: T2 reads an object that has been modified by T1, but T1 has not committed.
- T1 transfers $100 from A to B. T2 adds 6% interests to A and B. A non-serializable schedule is:
  - Step 1: deduct $100 from A.
  - Step 2: add 6% interest to A & B.
  - Step 3: credit $100 in B.
- Why is the problem?
  - The result is different from any serial schedule -> Bank adds $6 less interest.
- A Transaction must leave DB in a consistent state after it completes!

T1	T2
R(A)	
W(A-100)	
	R(A)
	W(A+6%)
	R(B)
	W(B+6%)
	Commit
R(B)	
W(B+100)	
Commit	

19

RW Conflicts (Unrepeatable Read)

- Situation: T2 modifies A that has been read by T1, while T1 is still in progress.
  - When T1 tries to read A again, it will get a different result, although it has not modified A in the meantime.
- What is the problem?
- A is the number of available copies of a book = 1. T1 wants to buy one copy. T2 wants to buy one copy. T1 gets an error.
  - The result is different from any serial schedule

T1	**T2**
R(A==1)	
Check if (A>0)	
	R(A==1)
	Check if (A>0)
	W(A=0)
W(A) **Error!**	
	Commit
Commit	

20

WW Conflict (Overwriting Uncommitted Data)

- T1 wants to set salaries of Harry & Larry to ($2000, $2000). T2 wants to set them to ($1000, $1000).
- What wrong results can the left schedule produce?
  - The left schedule can produce the results ($1000, $2000).
  - The result is different from any serial schedule.
  - This is called lost update (T2 overwrites T1's A value, so T1's value of A is lost.)

T1	T2
W(A=2000)	
	W(A=1000)
	W(B=1000)
	Commit
W(B=2000)	
Commit	

21

Schedules with Aborted Transactions

- Serializable schedule needs to produce the correct results under aborted transactions.
  - For aborted transactions, undo all their actions as if they were never carried out (atomic property).
- What is the problem?
  - Undo T1 => T2 has read a value for A that should never been there. But T2 has committed! (may not be able to undo committed actions).
  - This is called unrecoverable schedule.

T1	T2
R(A)	
W(A-100)	
	R(A)
	W(A+6%)
	R(B)
	W(B+6%)
	Commit
Abort	

22

Another Problem in Undo

- Undo T1: restore value of A to before T1's change (A=5).
- What is the Problem?
  - T2's change to A is also lost, even if T2 has already committed.

T1	T2
A=5	
R(A)	
W(A) // A=6	
	R(A)
	W(A) // A=7
	R(B)
	W(B)
	Commit
Abort	

23

Lock-Based Concurrency Control

- Concurrency control ensures that
  - (1) Only serializable, recoverable schedules are allowed
  - (2) Actions of committed transactions are not lost while undoing aborted transactions.
- How to guarantee safe interleaving of transactions' actions (serializability & recoverability)?

T1	T2
A=5	
R(A)	
W(A) // A=6	
	R(A)
	W(A) // A=7
	R(B)
	W(B)
	Commit
Abort	

24

Lock-Based Concurrency Control

- Strict Two-Phase Locking (Strict 2PL)
- A lock is a small bookkeeping object associated with a DB object.
  - Shared lock: several transactions can have shared locks on the same DB object.
  - Concurrent read? Concurrent write?
  - Exclusive lock: only one transaction can have an exclusive lock on a DB object.
  - Concurrent read? Concurrent write?

25

Strict 2PL

- Rule #1: If a transaction T wants to read an object, it requests a shared lock. Denote as S(O).
- Rule #2: If a transaction T wants to write an object, it requests an exclusive lock. Denote as X(O).
- Rule #3: When a transaction is completed (aborted), it releases all held locks.

26

Strict 2PL

- What happens when a transaction cannot obtain a lock?
  - It suspends
- Why shared lock for read?
  - Avoid RW/WR conflicts
- Why exclusive lock for write?
  - Avoid RW/WR/WW conflicts
- Requests to acquire or release locks can be automatically inserted into transactions.

27

Example #1 of Strict 2PL

T1
S(A)
R(A)
X(C)
R(C)
W(C)
Commit

T2
S(A)
R(A)
X(B)
R(B)
W(B)
Commit

T1	T2
	t
t	

28

Example #2 of Strict 2PL

T1
X(A)
R(A)
W(A)
X(B)
R(B)
W(B)
Commit // release locks

T2
X(A)
R(A)
W(A)
X(B)
R(B)
W(B)
Commit // release locks

T1	T2

29

"Strict" 2PL seems too strict?

possible for better concurrency?

T1
X(A)
R(A)
W(A)
X(B)
R(B)
W(B)
Commit // release locks

T2
X(A)
R(A)
W(A)
X(B)
R(B)
W(B)
Commit // release locks

T1	T2
X(A)	Suspend
R(A)	
W(A)	
X(B)	
R(B)	
W(B)	
Commit // release locks	
	X(A)
	R(A)
	W(A)
	X(B)
	R(B)
	W(B)
	Commit // release locks

T1	T2
X(A)	Suspend
R(A)	
W(A)	
X(B)	
R(B)	
W(B)	
Release X(A)	
Release X(B)	
Commit	
	X(A)
	R(A)
	W(A)
	R(B)
	W(B)
	Commit // release locks

30

2PL: without strict

What is the tradeoff for better concurrency?

T1	T2
X(A)	X(A)
R(A)	R(A)
W(A)	W(A)
X(B)	X(B)
R(B)	R(B)
W(B)	W(B)
Commit // release locks	Commit // release locks

T1	T2
X(A)	Suspend
R(A)	
W(A)	
X(B)	
Release X(A)	
	X(A)
	R(A)
	W(A)
R(B)	
W(B)	
Release X(B)	
Commit	
	X(B)
	R(B)
	W(B)
	Release X(A)
	Release X(B)
	Commit

31

2PL: what if T1 aborts?

T1	T2
X(A)	X(A)
R(A)	R(A)
W(A)	W(A)
X(B)	X(B)
R(B)	R(B)
W(B)	W(B)
Commit // release locks	Commit // release locks

T1	T2
X(A)	Suspend
R(A)	
W(A)	
X(B)	
Release X(A)	
	X(A)
	R(A)
	W(A)
R(B)	
W(B) // abort!	
Release X(B)	
Commit	
	X(B)
	R(B)
	W(B)
	Release X(A)
	Release X(B)
	Commit

32

Try "Strict" 2PL & Interleaving Executin

T1
X(A)
R(A)
W(A)
X(B)
R(B)
W(B)
Commit // release locks

T2
X(B)
R(B)
W(B)
X(A)
R(A)
W(A)
Commit // release locks

T1	T2

33

Deadlocks

- T1 and T2 will make no further progress.
- Two ways to handle deadlock:
  - Prevent deadlocks from occurring.
  - Detection deadlocks and resolve them.
- Detect deadlocked transactions by timeout (assuming they are waiting for a lock for too long) Abort the transactions.

T1	T2
X(A)	
	X(B)
Request X(B) Blocked!	
	Request X(A) Blocked too!

34

Deadlocks

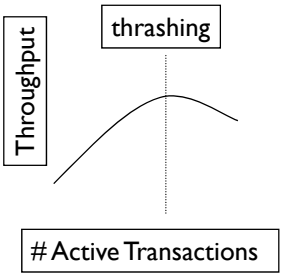
- T1 and T2 will make no further progress.
- How to handle deadlock (two ways)?
  - Prevent deadlocks from occurring.
  - Detect deadlocks and resolve them.
- How to do deadlock prevention and detection?

T1	T2
X(A)	
	X(B)
Request X(B) Blocked!	
	Request X(A) Blocked too!

35

Performance of Locking

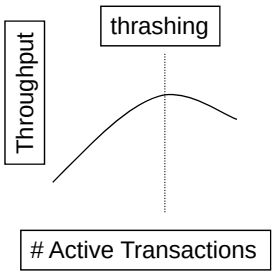
- Lock-based schemes are designed to resolve conflicts between transactions. It has two mechanisms:
  - Blocking: waiting for a lock.
  - Aborting: waiting for too long, restarting it.
- Both have costs and may impact throughput (# transactions completed per second).
- Very common system graph on the left



36

Performance of Locking

- More active transaction executing concurrently
  - Potentially higher concurrency gain
  - And higher the probably of blocking.
- Thrashing can occur when too many blocked transactions (or aborted transactions).
- How to guard against thrashing?



37

Improve Throughput

- Prevent thrashing: monitor % blocked transactions and reduce the number of active transactions executing concurrently.
- Other methods to improve throughputs:
  - Lock the smallest sized objects possible (reduce the likelihood of two transactions need the same lock).
  - Reduce the time that transaction hold locks (reduce blocking time of other transactions)
  - Reduce hot spots (hot spots = frequently accessed and modified objects).

38

What to Lock in SQL?

- Option 1: table granularity
  - T1: shared lock on S
  - T2: exclusive lock on S
  - Big sized object -> low concurrency (T1 no|| T2)
- Option 2: row granularity
  - T1: shared lock on rows with rating = 8.
  - T2: exclusive lock on rows with S.name="Joe" AND S.rating=8
  - Smaller granularity -> better concurrency (T1 || T2)

```
<T1>
SELECT S.rating, MIN(S.age)
FROM    Sailors S
WHERE   S.rating = 8

<T2>
UPDATE Sailors S
SETS.age=10
WHERE   S.name="Joe" AND
    S.rating=8
```

39