Transactions

Introduction to Databases

CompSci 316 Fall 2020

DUKE
COMPUTER SCIENCE

Announcements (Thu. Oct 29)

- Gradiance4—XML, due next Thursday (11/5).
  - Count() returns the number of elements returned by a Xpath
  - Execute your Xpath query and see how many elements at the outermost level are returned (not the nested elements)
- LectureQuiz-4-ACID (today after class) due on Thursday 11/5
- HW7-MongoDB/JSON posted, due Tuesday 11/10
  - To be done in project group, no collaboration outside project grp
  - One submission per project group to gradescope
  - Set up a common time, work on it together!
  - You need to know JSON/MongoDB only for this HW, not included in Final exam (XML/Lec 9 is included in Final)
- No other written/programming homework!
- Gradiance Quizzes on Transactions due on Thursday 11/12
  - Try to solve early when a quiz is posted
- Final project submission due Monday 11/16 (LDOC)
  - See project doc file on what to submit

Mark your calendars for the HW deadlines!

- Tuesday Nov 3 – Election day
  - Class on – watch the video later if you cannot attend-- Lecture (Transaction Concurrency Control/Recovery) included in Final – Gradiance quiz deadline moved

Where are we now?

Relational model and queries
- Relational Model
- Query in SQL
- Query in RA

Database Design
- E/R diagram (design from scratch)
- Normal Forms (refine design)

DBMS Internals and Query Processing
- Storage
- Index
- Join algo/Sorting
- Execution/ Optimization

Beyond Relational Model
- XML
- NOSQL **JSON/MongoDB**

Transactions
- Basics
- Concurrency Control
- Recovery

(Basic) Big Data Processing
- Map-Reduce
- Parallel DBMS

Covered

To be covered

Next

So far: One query/update
One machine

Multiple query/updates
One machine

Transactions

One query/update
Multiple machines

Parallel query processing
Map-Reduce, Spark, ..
Distributed query processing

Multiple query/updates, multiple machines:
Distributed transactions, Two-Phase Commit protocol, .. (not covered)

Why should we care about running multiple queries/updates/programs on a machine concurrently?

Motivation: Concurrent Execution

- Concurrent execution of user programs is essential for good DBMS performance.
  - Disk accesses are frequent, and relatively slow
  - it is important to keep the CPU busy by working on several user programs concurrently
  - short transactions may finish early if interleaved with long ones
- May increase system throughput (avg. #transactions per unit time)
- May decrease response time (avg. time to complete a transaction)

Transactions

```
T1:    BEGIN   A=A+100,   B=B-100   END
T2:    BEGIN   A=1.06*A,   B=1.06*B   END
```

- A transaction is the DBMS's abstract view of a user program
- a sequence of reads and write
    - DBMS only cares about R/W of "elements" (tuples, tables, etc)
- the same program executed multiple times would be considered as different transactions

Example

- Consider two transactions:

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

- Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.
- However, the net effect *must* be equivalent to these two transactions running serially in some order

Are these interleaving (schedule) good?

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

- Schedule 1:

T1:	A=A+100,		B=B-100	
T2:		A=1.06*A,		B=1.06*B

- Schedule 2:

T1:	A=A+100,		B=B-100
T2:		A=1.06*A, B=1.06*B	

- Schedule 3:

T1:		A=A+100,		B=B-100
T2:	A=1.06*A,		B=1.06*B	

Example: View of DBMS

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

- Schedule 2:

T1:	A=A+100,		B=B-100
T2:		A=1.06*A, B=1.06*B	

❖ The DBMS's view (and Notations!):

T1:	R(A), W(A),		R(B), W(B)
T2:		R(A), W(A), R(B), W(B)	

$R_1(A), W_1(A), R_2(A), W_2(A), R_2(B), W_2(B), R_1(B), W_1(B)$

C_1 = "Commit" by Transaction T1.
A_1 = "Abort" by Transaction T1

(next slide)

- Two possible representation of schedules
- No message passing
- Fixed set of objects (for now)

Commit and Abort

```
T1:     BEGIN   A=A+100,   B=B-100   END
T2:     BEGIN   A=1.06*A,   B=1.06*B   END
```

- A transaction might commit after completing all its actions
- or it could abort (or be aborted by the DBMS) after executing some actions

Concurrency Control and Recovery

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

- Concurrency Control
  - (Multiple) users submit (multiple) transactions
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions
  - user should think of each transaction as executing by itself one-at-a-time
  - The DBMS needs to handle concurrent executions
- Recovery
  - Due to crashes, there can be "partial" transactions
    - DBMS needs to ensure that they are not visible to other transactions
  - Also there can be some "completed" transactions with updated data still in memory (not yet to disk) and therefore lost in a crash
    - DBMS needs to ensure that the updates eventually go to disk

ACID Properties

- Atomicity
- Consistency
- Isolation
- Durability

Recall our
Disk-memory diagram!

Atomicity

```
T1:    BEGIN   A=A+100,   B=B-100   END
T2:    BEGIN   A=1.06*A,   B=1.06*B   END
```

- A user can think of a transaction as always executing all its actions in one step, or not executing any actions at all
  - Users do not have to worry about the effect of incomplete transactions

Transactions can be aborted (terminated) by the DBMS or by itself

- because of some anomalies during execution (and then restarts)
- the system may crash (say no power supply)
- may decide to abort itself encountering an unexpected situation
  e.g. read an unexpected data value or unable to access disks

Ensured by recovery methods using "Logs" by "undo"-ing incomplete tr.

Consistency

```
T1:      BEGIN   A=A+100,   B=B-100   END
T2:      BEGIN   A=1.06*A,   B=1.06*B   END
```

- Each transaction, when run by itself with no concurrent execution of other actions, must preserve the consistency of the database
  - e.g. if you transfer money from the savings account to the checking account, the total amount still remains the same

Responsibility of programmer's code and ensured by DBMS through other properties

Isolation

```
T1:     BEGIN   A=A+100,   B=B-100   END
T2:     BEGIN   A=1.06*A,   B=1.06*B   END
```

- A user should be able to understand a transaction without considering the effect of any other concurrently running transaction
  - even if the DBMS interleaves their actions
  - transaction are "isolated or protected" from other transactions

Often ensured by "Locks",
and other concurrency control approaches

Durability

End of lecture on 10/29

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

- Once the DBMS informs the user that a transaction has been successfully completed, its effect should persist
  - even if the system crashes before all its changes are reflected on disk

Ensured by recovery methods using "Logs" by "redo"-ing complete/committed tr.

Announcements (Tue. Nov 3)

- Today's attendance goes to everyone

Deadlines:

- Thursday 11/5:
  - (1) Gradiance4—XML due
  - (2) LectureQuiz-4-ACID due
- Tuesday 11/10
  - HW7-MongoDB/JSON due
  - One submission per project group to gradescope, no collaboration outside project group
  - You need to know JSON/MongoDB only for this HW, not included in Final exam
- Thursday 11/12
  - Two Gradiance Quizzes on Transactions due
  - To be released on Thursday 11/5
- Monday 11/16 (LDOC)
  - Final project submission due

Schedule

- An actual or potential sequence for executing actions as seen by the DBMS

- A list of actions from a set of transactions
  - includes READ, WRITE, ABORT, COMMIT

- Two actions from the same transaction T MUST appear in the schedule in the same order that they appear in T
  - cannot reorder actions from a given transaction

Scheduling Transactions

- Serial schedule: Schedule that does not interleave the actions of different transactions
- Equivalent schedules: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule
- Serializable schedule: A schedule that is equivalent to some serial execution of the committed transactions
  - Note: If each transaction preserves consistency, every serializable schedule preserves consistency

Serial Schedule

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

- If the actions of different transactions are not interleaved
  - transactions are executed from start to finish one by one

- Simple, but advantages of concurrent execution lost

Serializable Schedule

- Equivalent to "some" serial schedule
- However, no guarantee on T1-> T2 or T2 -> T1

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

serial schedule

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

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

serializable schedules

(Later, how to check for serializability)

Anomalies with Interleaved Execution

- Conflicts may arise if one transaction wants to write to a data that another transaction reads/writes

- Write-Read (WR) – reading uncommitted or "dirty" data
- Read-Write (RW) – unrepeatable reads
- Write-Write (WW) – overwriting uncommitted data or "lost updates"

- No conflict with RR if no write is involved

SQL transactions

- A transaction is automatically started when a user executes an SQL statement
- Subsequent statements in the same session are executed as part of this transaction
  - Statements see changes made by earlier ones in the same transaction
  - Statements in other concurrently running transactions do not
- COMMIT command commits the transaction
  - Its effects are made final and visible to subsequent transactions
- ROLLBACK command aborts the transaction
  - Its effects are undone

Fine prints

- Schema operations (e.g., CREATE TABLE) implicitly commit the current transaction
- Many DBMS support an AUTOCOMMIT feature, which automatically commits every single statement
    - You can turn it on/off through the API

SQL isolation levels

- Strongest isolation level: SERIALIZABLE
  - Complete isolation
- Weaker isolation levels:
- REPEATABLE READ,
- READ COMMITTED,
- READ UNCOMMITTED
- Increase performance by eliminating overhead and allowing higher degrees of concurrency
- Trade-off: sometimes you get the “wrong” answer

READ UNCOMMITTED

- Can read "dirty" data (WR conflict)
  - A data item is dirty if it is written by an uncommitted transaction
- Problem: What if the transaction that wrote the dirty data eventually aborts?
- Example: wrong average
  - -- T1:

```
-- T1:                              -- T2:
UPDATE User
SET pop = 0.99
WHERE uid = 142;
                                    SELECT AVG(pop)
                                    FROM User;
 ROLLBACK;
                                    COMMIT;
```

READ COMMITTED

- No dirty reads, but non-repeatable reads possible (RW conflicts)
  - Reading the same data item twice can produce different results
- Example: different averages

```
-- T1:                          -- T2:
                                SELECT AVG(pop)
                                FROM User;
UPDATE User
SET pop = 0.99
WHERE uid = 142;
COMMIT;
                                SELECT AVG(pop)
                                FROM User;
                                COMMIT;
```

REPEATABLE READ

- Reads are repeatable, but may see phantoms
- Example: different average (still!)
  - -- T1:

```
-- T1:                                   -- T2:
                                         SELECT AVG(pop)
                                         FROM User;
INSERT INTO User
VALUES(789, 'Nelson',
     10, 0.1);
COMMIT;
                                         SELECT AVG(pop)
                                         FROM User;
                                         COMMIT;
```

Summary of SQL isolation levels

Isolation level/anomaly	Dirty reads	Non-repeatable reads	Phantoms
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Impossible	Possible	Possible
REPEATABLE READ	Impossible	Impossible	Possible
SERIALIZABLE	Impossible	Impossible	Impossible

- Syntax: At the beginning of a transaction,
  SET TRANSACTION ISOLATION LEVEL *isolation_level*
  [READ ONLY | READ WRITE];
  - READ UNCOMMITTED can only be READ ONLY

- PostgreSQL defaults to READ COMMITTED

Bottom line

- Group reads and dependent writes into a transaction in your applications
  - E.g., enrolling a class, booking a ticket
- Anything less than SERIALABLE is potentially very dangerous
  - Use only when performance is critical
  - READ ONLY makes weaker isolation levels a bit safer

Conflicting operations

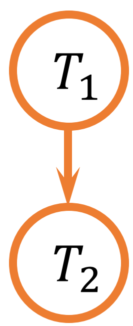
- Two operations on the same data item conflict if at least one of the operations is a write
  - r(X) and w(X) conflict
  - w(X) and r(X) conflict
  - w(X) and w(X) conflict
  - r(X) and r(X) do not conflict
  - r/w(X) and r/w(Y) do not conflict

- Order of conflicting operations matters
  - E.g., if T_1.r(A) precedes T_2.w(A), then conceptually, T_1 should precede T_2

Precedence graph

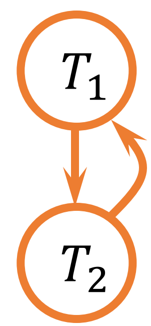
- A node for each transaction
- A directed edge from T_i to T_j if an operation of T_i precedes and conflicts with an operation of T_j in the schedule

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



Good:
no cycle

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



Bad:
cycle

Conflict-serializable schedule

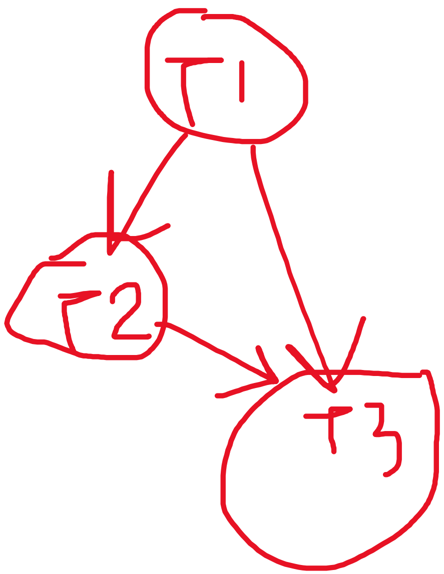
- A schedule is conflict-serializable iff its precedence graph has no cycles

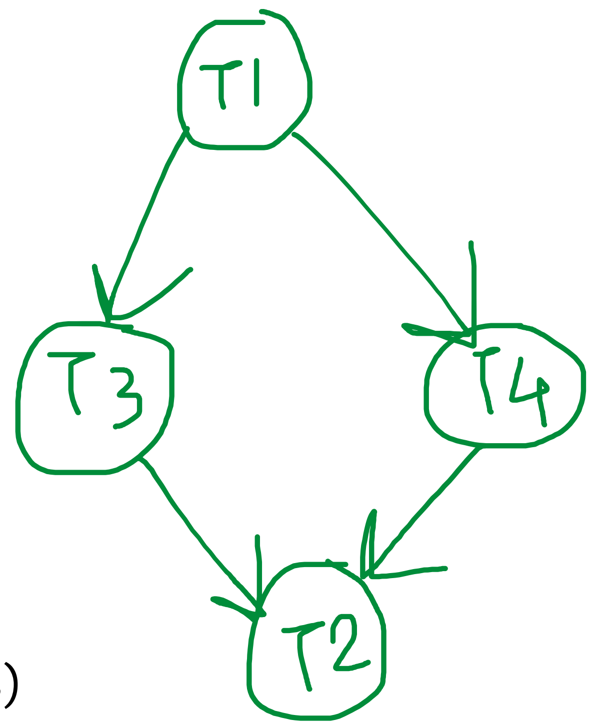
- A conflict-serializable schedule is equivalent to some serial schedule (and therefore is “good”)
  - In that serial schedule, transactions are executed in the “topological order” of the precedence graph (see next slide)
  - You can get to that serial schedule by repeatedly swapping adjacent, non-conflicting operations from different transactions (see next to next slide)

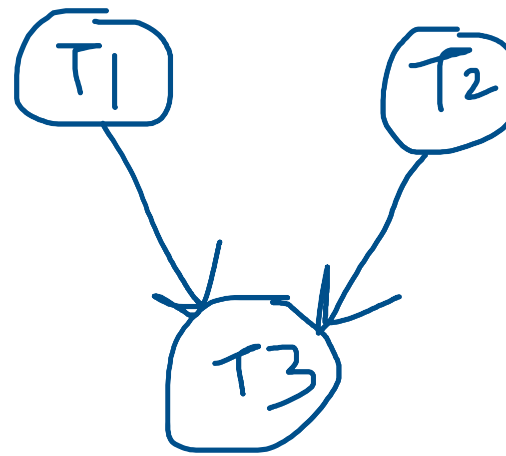
Topological order to find equivalent serial schedule(s)

End of lecture on 11/3

- List a node only after all its predecessors (nodes having a directed path to this node) are processed







Equivalent serial schedule (s)

T1, T2, T3

T1, T3, T4, T2

OR T1, T4, T3, T2

T1, T2, T3

OR T2, T1, T3

Swapping adjacent non-conflicting actions to reach an equivalent serial schedule

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

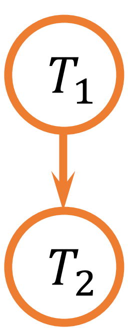
T_1	T_2
r(A)	
w(A)	
	r(A)
	w(A)
r(B)	
w(B)	
	r(C)
	w(C)

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

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

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

SERIAL



Good: no cycle

Locking (for Conurrency Control)

- 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	X
S	Yes	No
X	No	No

Grant the lock?

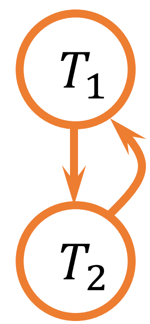
Compatibility matrix

Basic locking is not enough

Add 1 to both A and B (preserve A=B)	T_1	T_2	Multiply both A and B by 2 (preserves A=B)
	lock-X(A)		
Read 100	r(A)		
Write 100+1	w(A)		
	unlock(A)		
		lock-X(A)	
		r(A)	Read 101
		w(A)	Write 101*2
		unlock(A)	
		lock-X(B)	
		r(B)	Read 100
		w(B)	Write 100*2
		unlock(B)	
	lock-X(B)		
Read 200	r(B)		
Write 200+1	w(B)		
	unlock(B)		

Possible schedule under locking

But still not conflict-serializable!



$A \neq B$!

Two-phase locking (2PL)

- All lock requests precede all unlock requests
  - Phase 1: obtain locks, phase 2: release locks

T_1	T_2
lock-X(A)	
r(A)	
w(A)	
lock-X(B)	
unlock(A)	
	lock-X(A)
	r(A)
	w(A)
	lock-X(B)
	r(B)
	w(B)
r(B)	
w(B)	
unlock(B)	

Cannot obtain the lock on B until T_1 unlocks

2PL guarantees a conflict-serializable schedule

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

Remaining problems 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
- *Avoids Cascading Rollback* = Each transaction reads only data written by committed transactions.

- Even worse, what if T_2 commits before T_1?
  - Schedule is not recoverable if the system crashes right after T_2 commits
- *Recoverable* = Each transaction commits after all transactions from which it has read has committed.

Strict 2PL

- Only release locks at commit/abort time
  - A writer will block all other readers until the writer commits or aborts

- Used in many commercial DBMS
  - Oracle is a notable exception

Isolation levels not based on locks?

Snapshot isolation in Oracle

- Based on multiversion concurrency control
  - Used in Oracle, PostgreSQL, MS SQL Server, etc.
  - Intuition: uses a "private snapshot" or "local copy"
  - If no conflict make global or abort

- More efficient than locks, but may lead to aborts
- Other methods: Timestamp-based