ICS 321 Fall 2011

Overview of Transaction Processing

Asst. Prof. Lipyeow Lim

Information & Computer Science Department

University of Hawaii at Manoa

Transactions in SQL

- After connection to a database, a transaction is automatically started
  - Different connections -> different transactions
- Within a connection, a transaction is ended by
  - **COMMIT** or **COMMIT WORK**
  - **ROLLBACK** (= “abort”)
- DBMS can also initiate rollback and return an error.
- **SAVEPOINT** <savepoint name>
- **ROLLBACK TO SAVEPOINT** <savepoint name>
  - Locks obtained after savepoint can be released after rollback to that savepoint
- Using savepoints vs sequence of transactions
  - Transaction rollback is to last transaction only

Isolation levels in SQL

- SQL supports 4 isolation levels

SQL Isolation Levels	DB2 Isolation Levels	Dirty read	Unrepeatable Read	Phantom
READ UNCOMMITTED	UNCOMMITTED READ (**UR**)	Maybe	Maybe	Maybe
READ COMMITTED	CURSOR STABILITY * (**CS**)	No	Maybe	Maybe
REPEATABLE READ	READ STABILITY (**RS**)	No	No	Maybe
SERIALIZABLE	REPEATABLE READ (**RR**)	No	No	No

```
SET TRANSACTION ISOLATION LEVEL   SERIALIZABLE
```

```
SELECT *
FROM Reserves
WHERE SID=100
WITH UR
```

Anomaly: Dirty Reads

- T1 reads uncommitted data from T2 which may abort

T1	T2
	A=A+100
A=1.06*A	
Commit	
	B=B-100
	Abort

A = 20

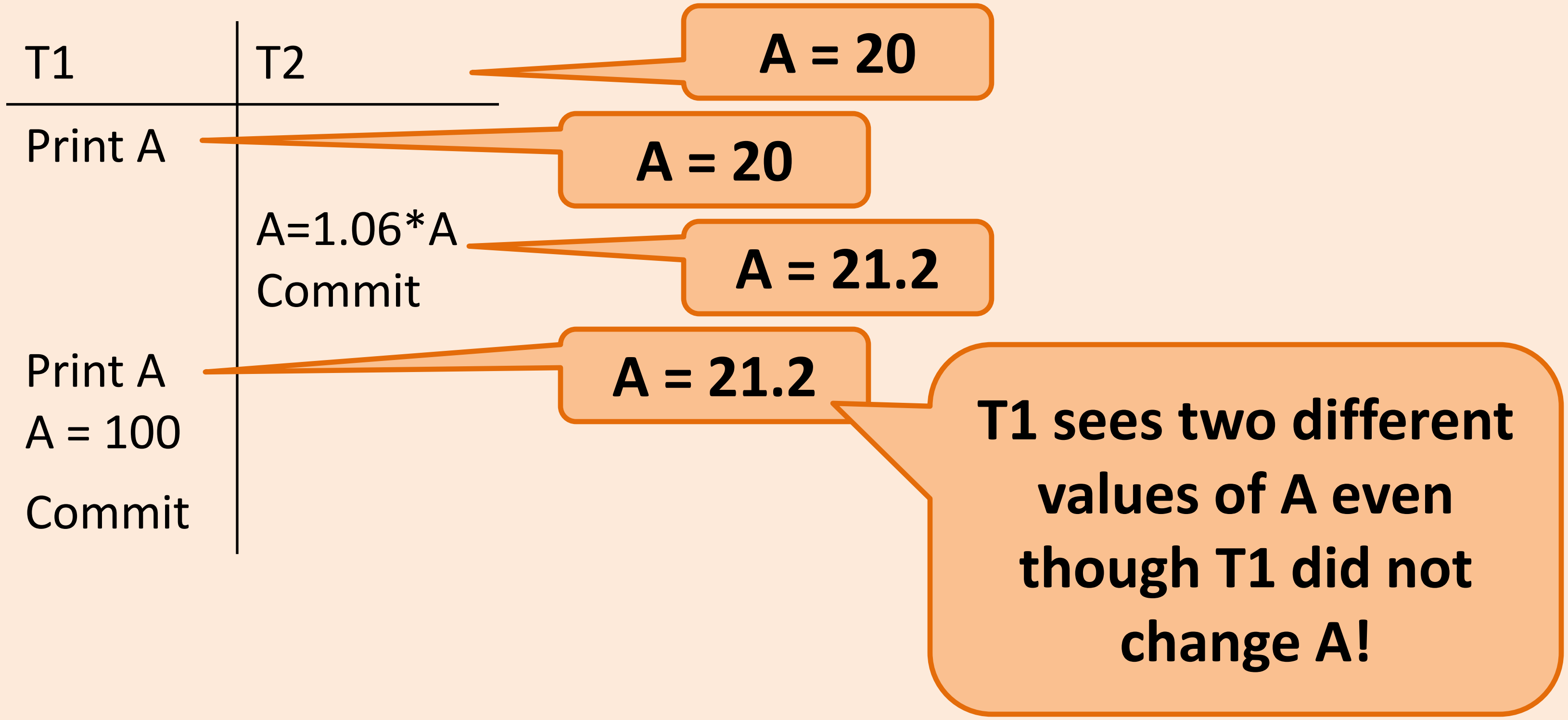
A = 120

A = 127.2

With T2 aborted correct value of A = 21.2

Anomaly: Unrepeatable Reads

- T1 sees two different values of A, because updates are committed from another transaction (T2)



T1	T2
Print A	
	A=1.06*A
	Commit
Print A	
A = 100	
Commit	

A = 20
A = 20
A = 21.2
A = 21.2
T1 sees two different values of A even though T1 did not change A!

Anomaly: Phantom Reads

- Multiple reads from the same transaction sees different set of tuples

T1	T2
Find all ics321 students	
	Enroll student D into ics321
	Commit
Find all ics321 students	
Commit	

{A,B,C}

Insert D

{A,B,C,D}

T1 sees two different results of the query even though T1 did not change the table!

Lock-based Concurrency Control

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
    - (Non-strict) 2PL Variant: Release locks anytime, but cannot acquire locks after releasing any lock.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only serializable schedules.
  - Additionally, it simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves more complex abort processing

Example (Strict 2PL)

- Consider the dirty read schedule

T1	T2
A=A+100	
	A=1.06*A
	Commit
B=B-100	
Abort	

A = 20

A = 120

Dirty read on A!

A = 127.2

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

With Strict 2PL, T2 can only access A when T1 aborts

Example (Non-Strict 2PL)

- Consider the dirty read schedule

T1	T2
A=A+100	
	A=1.06*A
	Commit
B=B-100	
Abort	

A = 20

A = 120

Dirty read on A!

A = 127.2

T1	T2
X(A)	
R(A)	
W(A)	
RX(A)	
	X(A)
	R(A)
	W(A)
	Commit
X(B)	
R(B)	
W(B)	
Abort	

With non-strict 2PL, T2 can still read uncommitted data if T1 aborts!

Deadlocks

- Cycle of transactions waiting for locks to be released
- DBMS has to either prevent or resolve deadlocks
- Common approach:
  - Detect via timeout
  - Resolve by aborting transactions

T1	T2
Req X(A)	**Req X(B)**
Gets X(A)	**Gets X(B)**
…	**….**
Req X(B)	
	Req X(A)

Aborting a Transaction

- If a transaction *T1* is aborted, all its actions have to be undone.
  - Not only that, if *T2* reads an object last written by *T1*, *T2* must be aborted as well!
- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
  - If *T1* writes an object, *T2* can read this only after *T1* commits.
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded.
  - This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up

Lock Granularity

- What should the DBMS lock ?
  - Row ?
  - Page ?
  - A Table ?

```
UPDATE  Sailors
SET     rating=0
WHERE   rating>9
```

```
SELECT  *
FROM    Sailors
```

```
SELECT  *
FROM    Sailors
WHERE rating < 2
```

```
UPDATE  Boats
SET     color=‘red’
WHERE   bid=13
```

```
UPDATE  Boats
SET     color=‘blue’
WHERE   bid=100
```

Crash Recovery

- **Transaction Manager**: DBMS component that controls execution (eg. managing locks).
- **Recovery Manager**: DBMS component for ensuring
  - Atomicity: undo actions of transactions that do not commit
  - Durability: committed transactions survive system crashed and media failures
- Assume atomic writes to disk.

The Log

- The following actions are recorded in the log:
  - *Ti writes an object*: the old value and the new value.
    - Log record must go to disk *before* the changed page! (Write Ahead Log property)
  - *Ti commits/aborts*: a log record indicating this action.
- Log records are chained together by Xact id, so it's easy to undo a specific Xact.
- Log is often *duplexed* and *archived* on stable storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.

Recovering from a Crash

- There are 3 phases in the *Aries* recovery algorithm:
  - *Analysis*: Scan the log forward (from the most recent *checkpoint*) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
  - *Redo*: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
  - *Undo*: The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)