Advanced Concurrency Control

Databases, Aarhus University

Ira Assent

Intended learning outcomes

- Be able to
  - Set isolation levels
  - Reason about and apply timestamp based protocols
  - Work with multiversion concurrency control techniques

ira@cs.au.dk

Recap: Locking

- Consider database state, i.e. reads and writes of transactions
  - $\mathbf{R_I(X)}$: T_I read_item(X), $\mathbf{W_I(X)}$: T_I write_item(X);
- Lock L an item X while in use, unlock when done
  - Exclusive lock: write_lock(X) to read or write – at most one transaction per item $\mathbf{LW_I(X)}$: T_I write_lock(X);
  - Shared lock: read_lock(X) to read – more than one transaction per item possible $\mathbf{LR_I(X)}$: T_I read_lock(X);
  - Unlocking to release item: $\mathbf{U_I(X)}$: T_I unlock(X);
- 2-phase protocol: growing phase requests locks, no lock release; shrinking phase: no lock requests, only releases
  - Generates precedence graphs without cycles → serializability
    - If T_j obtains lock on item previously locked by T_i, then T_i shrinking: can no longer acquire locks
    - Thus, implicit ordering of transactions T_i and T_j with respect to their phases, cannot close cycle
  - Strict 2-phase locking: write locks not released until commit time
  - Rigorous 2-phase locking: all locks not released until commit time

Basic 2PL locking: can T_2 lock B?

A. T_2 is granted a lock because T_1 is done writing to A
B. T_2 is granted a lock because the lock on B is a shared lock
C. T_2 is not granted a lock because T_1 holds a shared lock on B
D. T_2 is not granted a lock because T_1 has entered the shrinking phase already
E. T_2 is granted a lock because it has not yet released any locks

T1	T2	A	B
$LW_1(A); R_1(A);$		25	25
$W_1(A); LR_1(B); U_1(A);$		125	
	$LW_2(A); R_2(A);$		
	$W_2(A);$	250	
	$LW_2(B)$?		
$R_1(B);$			
$U_1(B);$			
	$U_2(A); R_2(B);$		
	$W_2(B); U_2(B)$		

$R_1(X)$: T_1 read_item(X), $W_1(X)$: T_1 write_item(X);

$LR_1(X)$: T_1 read_lock(X); $LW_1(X)$: T_1 write_lock(X); $U_1(X)$: T_1 unlock(X);

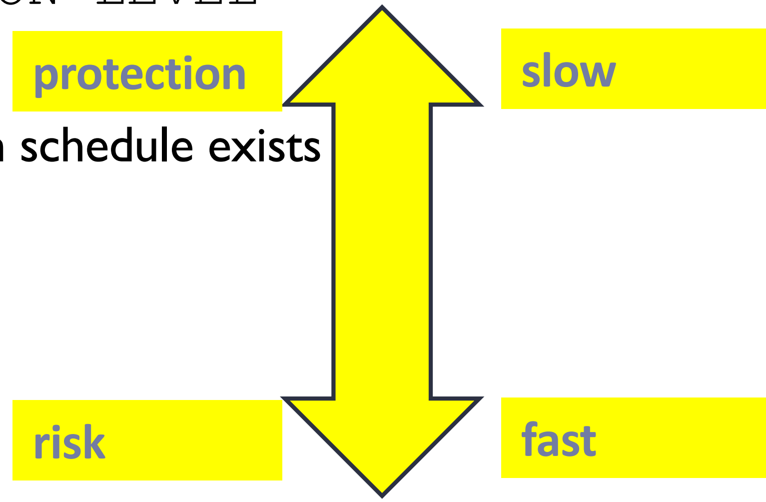
ira@cs.au.dk

Isolation Levels

- SQL defines four *isolation levels*
  - choices about acceptable kind of interference by other transactions
- Within a transaction, we can say:

```
SET TRANSACTION ISOLATION LEVEL
```

1. `SERIALIZABLE`
   - Equivalent serial execution schedule exists
   - Fulfills ACID requirements
2. `REPEATABLE READ`
3. `READ COMMITTED`
4. `READ UNCOMMITTED`



- Picking the right one depends on the application

https://dev.mysql.com/doc/refman/8.3/en/set-transaction.html#set-transaction-isolation-level

ira@cs.au.dk

Isolation level effects

- **`REPEATABLE READ`** is weaker than `SERIALIZABLE`
  - Read and write locks on rows, but not on range
  - If a row is read again then it has the same value
    - Read locks are set on the rows that have been read
  - No dirty reads, but more rows may appear! (because no lock set on range in which new rows could appear) (*phantom reads*)
- **`READ COMMITTED`** is weaker than `REPEATABLE READ`
  - Read and write locks on rows, but read locks are released immediately after reading
  - No dirty reads, but not necessarily the same value every time!
- **`READ UNCOMMITTED`** is weaker than `READ COMMITTED`
  - No read locks
  - Dirty reads are possible!
    - If a row is read again, then it may suddenly have a different value!
    - Rows may appear and disappear!

ira@cs.au.dk

Joe runs `serializable`, Sally `read uncommitted`

A. It may be that Sally sees no prices for Joe's bar
B. Sally will see prices from before or after all of Joe's changes
C. Sally will see the prices as they were before Joe runs
D. Sally will see the prices as they are after Joe runs

```
Sally
(max)    SELECT MAX(price) FROM Sells
                    WHERE bar = 'J Bar';
(min)    SELECT MIN(price) FROM Sells
                    WHERE bar = 'J Bar';
Joe
(del)    DELETE FROM Sells
         WHERE bar = 'J Bar';
(ins)    INSERT INTO Sells
         VALUES('J Bar', 'Heineken', 3.50);
```


ira@cs.au.dk

Isolation Level is Personal Choice

- Your choice, e.g., run serializable, affects only how **you** see the database, not how others see it
- Example: If Joe runs `serializable`, but Sally uses `read uncommitted`, then Sally might see no prices for Joe's Bar
  - i.e., it looks to Sally as if she ran in the middle of Joe's transaction

```
Sally
(max)    SELECT MAX(price) FROM Sells
                   WHERE bar = 'J Bar';
(min)    SELECT MIN(price) FROM Sells
                   WHERE bar = 'J Bar';
Joe
(del)    DELETE FROM Sells
         WHERE bar = 'J Bar';
(ins)    INSERT INTO Sells
         VALUES('J Bar', 'Heineken', 3.50);
```


ira@cs.au.dk

Isolation level serializable

- If Sally = (max)(min) and Joe = (del)(ins) are each transactions, and Sally runs with isolation level `SERIALIZABLE`, then she will see the database either before or after Joe runs, but not in the middle
  - "standard" ACID behavior
- It is up to the DBMS on how to implement this, e.g.:
  - True isolation in time
  - Keep Joe's old prices around to answer Sally's queries
  - Locking
  - …

```
Sally
(max)     SELECT MAX(price)
          FROM Sells
          WHERE bar = 'J Bar';
(min)     SELECT MIN(price)
          FROM Sells
          WHERE bar = 'J Bar';
Joe
(del)     DELETE FROM Sells
          WHERE bar = 'J Bar';
(ins)     INSERT INTO Sells
          VALUES('J Bar',
          'Heineken', 3.50);
```


ira@cs.au.dk

Example isolation levels

- If Sally runs `READ COMMITTED`, she can see only committed data, but not necessarily the same data each time
  - Example: Under `READ COMMITTED`, the interleaving (max)(del)(ins)(min) is allowed, as long as Joe commits
    - Sally sees MAX < MIN
- Suppose Sally runs `REPEATABLE READ`, and order of execution is (max)(del)(ins)(min)
  - (max) sees prices 2.50 and 3.00
  - (min) can see 3.50, but must also see 2.50 and 3.00, because they were seen on the earlier read by (max)
- If Sally runs `READ UNCOMMITTED`, she could see a price 3.50 even if Joe later aborts (dirty read)

```
Sally
(max)    SELECT MAX(price)
         FROM Sells
         WHERE bar = 'J Bar';
(min)    SELECT MIN(price)
         FROM Sells
         WHERE bar = 'J Bar';
Joe
(del)    DELETE FROM Sells
         WHERE bar = 'J Bar';
(ins)    INSERT INTO Sells
         VALUES('J Bar',
         'Heineken', 3.50);
```


ira@cs.au.dk

Read-Only Transactions

- Transactions that only read can never violate serializability

- This can be declared in SQL:

  `SET TRANSACTION READ ONLY;`

  (a DBMS generally cannot infer which statements a transaction will execute ahead of time)

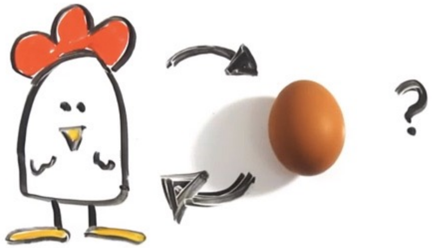
- Other transactions need never wait for such a transaction to finish

ira@cs.au.dk

Transactions and Constraints

- Constraints are checked after modifications
- If violated, the transaction is rolled back

- Sometimes, a transaction must temporarily violate a constraint




```
CREATE TABLE Chicken (chickenID INT PRIMARY KEY,
                      eggID INT REFERENCES Egg(eggID));
CREATE TABLE Egg(eggID INT PRIMARY KEY,
                 chickenID INT REFERENCES Chicken(chickenID));
```

- A constraint can be declared deferrable by `DEFERRABLE INITIALLY DEFERRED`
  - It is then only checked at commit time
  - Not available in MySQL (but e.g. PostgreSQL)

ira@cs.au.dk

The Timestamp Protocol

- Manage concurrency as equivalent to serial execution in timestamp order
  - Each transaction is given a timestamp when entering system
  - Timestamps are an increasing integer sequence
  - Access to a data item must be in timestamp order
    - Idea: as long as transactions work on data in the order they would have seen it if they just proceeded serially, all good
  - **No locking!**

- Protocol maintains for each item X two values
  - **read_TS(X)** : the timestamp of latest transaction to read item
  - **write_TS(X)**: the timestamp of latest transaction to write item

- Avoid confusion: the **latest** transaction to have read/written the item, is the **youngest** transaction, is the one with the **largest** timestamp

ira@cs.au.dk

Implementing the Timestamp Protocol

- Permit access that follows the timestamp order
  - When T reads X, write_TS(X) must be earlier than timestamp of T
  - When T writes X, write_TS(X) **and** read_TS(X) must be earlier
- Abort T if out of order, and restart with a new (later) timestamp

- Let T be the timestamp of the executing transaction

read_item(X): **if** T ≥ write_TS (X)
        perform read
        read_TS(X) ← max(T, read_TS(X))
    else abort

write_item(X): **if** T ≥ read_TS(X) **and** T ≥ write_TS(X)
        perform write
        write_TS (X) ← T
    else
        if T < read_TS(X)
            abort
        else do nothing

read_TS(X) ≤ T < write_TS (X) – so already overwritten…
(Thomas' rule)

Timestamp Protocol Example

- Transactions with timestamps 1 through 5

T_1	T_2	T_3	T_4	T_5
				$R_5(X)$
	$R_2(Y)$			
$R_1(Y)$				
		$W_3(Y)$		
		$W_3(Z)$		
				$R_5(Z)$
	$R_2(Z)$			
$R_1(X)$				
			$W_4(Z)$	
				$W_5(X)$
				$W_5(Z)$

read_ TS(x)	write_ TS(x)	read_ TS(y)	write_ TS(y)	read_ TS(z)	write_ TS(z)

ira@cs.au.dk

Timestamp Protocol Example

- Transactions with timestamps 1 through 5

T_1	T_2	T_3	T_4	T_5	read_TS(x)	write_TS(x)	read_TS(y)	write_TS(y)	read_TS(z)	write_TS(z)
				$R_5(X)$	5					
	$R_2(Y)$						2			
$R_1(Y)$							2			
		$W_3(Y)$						3		
		$W_3(Z)$								3
				$R_5(Z)$					5	
	$R_2(Z)$ abort									
$R_1(X)$					5					
			$W_4(Z)$ abort							
				$W_5(X)$		5				
				$W_5(Z)$						5

You cannot see the future

You cannot change the past

ira@cs.au.dk

Schedules

A. Both schedules allowed by locks but not timestamps.
B. Both schedules allowed by timestamps but not by locks.
C. Both schedules allowed by locks and timestamps.
D. Both schedules not allowed by locks and timestamps.
E. Top schedule allowed by locks but not timestamps, bottom schedule allowed by timestamps but not by locks.
F. Bottom schedule allowed by locks but not timestamps, top schedule allowed by timestamps but not by locks.

T_1	T_2
$R_1(A)$	
	$R_2(B)$
	$W_2(B)$
$R_1(B)$	

T_1	T_2	T_3
$R_1(A)$		
	$R_2(A)$	
		$R_3(D)$
		$W_3(D)$
		$W_3(A)$
	$R_2(C)$	
$W_1(B)$		
	$W_2(B)$	

ira@cs.au.dk

Cascading Rollbacks

- One transaction aborting can cause other transactions to abort
  - T_1 aborts $\Rightarrow$ have to rollback T_2 and T_3
- How to eliminate these cascading rollbacks?
  - Don't let transactions read "dirty" uncommitted data
    - Benefit of e.g. strict 2PL – dirty reads not possible
    - No reading of data written by transaction that is not yet committed

T_1	T_2	T_3
$LW_1(X)$		
$W_1(X)$		
$LW_1(Y)$		
$U_1(X)$		
	$LW_2(X)$	
	$R_2(X)$	
	$W_2(X)$	
	$U_2(X)$	
…		$LW_3(X)$

$R_1(X)$: T_1 read_item(X), $W_1(X)$: T_1 write_item(X);

$LR_1(X)$: T_1 read_lock(X); $LW_1(X)$: T_1 write_lock(X); $U_1(X)$: T_1 unlock(X);

ira@cs.au.dk

Strict Timestamp Based Concurrency Control

- How to avoid cascading rollbacks?
  - Transactions should read only committed values.
- **Strict** timestamp concurrency control protocol

read X: **if** T > write_TS(X)
  read_TS(X) ← max(T, read_TS(X))
  wait for a committed value of X
  perform read
else abort

write X: **if** T ≥ read_TS(X) **and** T ≥ write_TS(X)
  write_TS(X) ← T
  wait until X value is a committed value and pending reads are done
  perform write
else
  if T < read_TS(X) abort
  else do nothing

ira@cs.au.dk

Terminology overview

- **Cascading rollback**
  - Uncommitted transactions that read an item from aborted transaction must be rolled back
    - Means those transactions worked with "non-existing" value → needs fixing by trying over
- **Cascadeless schedule**
  - Every transaction reads only items written by committed transactions
    - Do not allow access to "non-existing" values
- **Strict Schedule**
  - Transaction can neither read or write item until the last transaction that wrote it has committed
- **Recoverable schedule**
  - No transaction commits until all transactions that have written an item that it read have committed
    - Make sure that we can fix any situation with "non-existing" value before anything is permanent!

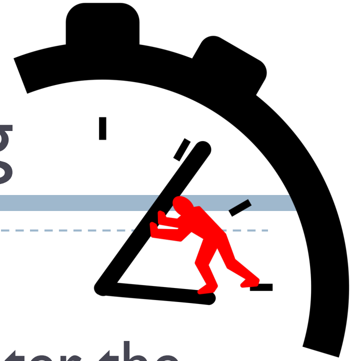
ira@cs.au.dk

Multiversion concurrency control

- Observation:
  - Read operations are sometimes rejected in concurrent processing, because an item has been written by another transaction

- Key idea:
  - Maintain older versions of data items
  - When reading, allocate the right version to the read operation of a transaction
    - Means reads never rejected
- Advantages and Disadvantages:
  - Significantly more storage (RAM and disk) is required to maintain multiple versions
  - To check unlimited growth of versions, a garbage collection is run when old versions no longer needed (induces some overhead)
  - In some cases, older versions available for recovery or for temporal databases (full history available: when was what valid?)

Multiversion based on timestamp ordering



- Timestamping
  - Grant access based on order in which transactions enter the system
  - In single version systems, permit access:
    - When *T* reads *X*, *write_TS*(*X*) must be earlier than timestamp *T*
    - When *T* writes *X*, *write_TS*(*X*) and *read_TS*(*X*) must be earlier
    - Abort *T* if out of order, and restart with a new (later) timestamp

  - Now, need a version that "fits" Ts own timestamp
    - When T reads a version of Q, that version must be written by a transaction with an earlier timestamp than that of T (and the oldest among all that qualify)
    - When T writes a version of Q, that version must not have been written or read by a transaction with a younger timestamp
      - Otherwise abort T if out of order (occurs only for write attempts)

ira@cs.au.dk

Multiversion timestamps

- Assume $X_1, X_2, \ldots, X_n$ are the versions of a data item X created by a write operation of transactions
  - With each X_i a read_TS (read timestamp) and a write_TS (write timestamp) are associated
- **read_TS(X_i)**: The read timestamp of X_i is the largest of all the timestamps of transactions that have successfully read version X_i (= most recent read)
- **write_TS(X_i)**: The write timestamp of X_i that wrote the value of version X_i (= the write that created this version)

➢ A new version of X_i is created only by a write operation

ira@cs.au.dk

Rules in multiversion timestamping

- To ensure serializability, the following two rules are used
- Rule 1: reject T if it attempts to overwrite a version (with the most recent timestamp that is still earlier than its own timestamp, so in the right creation order) if that was already read by a younger T' (meaning that T' would then have read out of order)
  1. If transaction T issues write_item(X) and version X_i has the highest write_TS(X_i) of all versions of X that is also less than or equal to TS(T), and read _TS(X_i) > TS(T), then abort and roll-back T; otherwise create a new version X_j and read_TS(X_j) = write_TS(X_j) = TS(T)
- Rule 2 guarantees that a read will never be rejected
  2. If transaction T issues read_item(X), find the version X_i that has the highest write_TS(X_i) of all versions of X that is also less than or equal to TS(T), then return the value of X_i to T, and set the value of read _TS(X_i) to the largest of TS(T) and the current read_TS(X_i)

ira@cs.au.dk

What happens in regular or multiversion timestamping?

A. T3 aborts in both.
B. T3 reads in both.
C. T3 aborts in regular only.
D. T3 reads in regular only.

T1	T2	T3	T4	A
150	200	175	225	RT=0
				WT=0
R1(A)				RT=150
W1(A)				WT=150
	R2(A)			RT=200
	W2(A)			WT=200
		R3(A)		?
			R4(A)	?

ira@cs.au.dk

Intended learning outcomes

- Be able to
  - Set isolation levels
  - Reason about and apply timestamp based protocols
  - Work with multiversion concurrency control techniques

Acknowledgements: includes slide material by Jeff Ullman, Torben Bach Pedersen, and teachcoop

ira@cs.au.dk

What was this all about?

Guidelines for your own review of today's session

- SQL Isolation Levels affect…
  - They can be controlled by…
- The timestamping protocol takes the approach of…
  - Cascading rollbacks are…
  - Recoverable schedules mean… and they guarantee that…
  - Cascadeless schedules are…
  - Strict schedules enforce … such that…
- Multiversion concurrency control is motivated by the observation that…
  - Multiversion means that the following information is maintained…
  - The rules applied based on this information are…

ira@cs.au.dk