CS5226 Lecture 8
Transaction Tuning II

Concurrency vs Consistency

Serializable Isolation Level

- Strict 2PL ensures conflict serializable schedules
- Poor performance - blocked / aborted transactions

Read Committed Isolation Level

- Better performance
- Vulnerable to lost update & unrepeatable read anomalies

Multiversion Concurrency Control (MVCC)

- Motivation: increase concurrency by not blocking reads
- Key idea: maintain multiple versions of each object
  - $W_i(O)$ creates a new version of object O
  - $R_i(O)$ reads the most recent version of O that is created by a committed Xact
- Advantages:
  - Readers are not blocked by writers
  - Readers do not block writers
  - Read-only Xacts are never aborted
- MVCC techniques:
  - Multiversion two-phase locking
  - Multiversion timestamp ordering
  - Snapshot isolation

Snapshot Isolation (SI)

- Widely used (e.g., Oracle, PostgreSQL, SQL Server, Sybase IQ)
- Each Xact T sees a snapshot of DB that consists of updates by Xacts that committed before T starts
- Each Xact T is associated with two timestamps:
  - start(T): the time that T starts
  - commit(T): the time that T commits

Concurrent Transactions

- Two Xacts T and T' are defined to be concurrent if they overlap
  - i.e., $[start(T), commit(T)] \cap [start(T'), commit(T')] \neq \emptyset$
- **Example**:

T_1	T_2	T_3
$R_1(B)$		
	$R_2(A)$	
$W_1(B)$		
$Commit_1$		
	$R_2(B)$	
	$W_2(A)$	
		$R_3(A)$
		$R_3(B)$
		$Commit_3$
	$Commit_2$	

Snapshot Isolation (SI)

- $\mathbf{W_i}(\mathbf{O})$ creates a version of O denoted by O_i
- $\mathbf{R_i}(\mathbf{O})$ reads the latest version of O that is created by a Xact that committed before T_i started;
  i.e., If $R_i(O)$ returns O_j, then
  1. $commit(T_j) < start(T_i)$, and
  2. For every Xact T_k, $k \neq j$, that has created a version O_k of O, if $commit(T_k) < start(T_i)$, then $commit(T_k) < commit(T_j)$
- Note: if T_i has updated O, then any $R_i(O)$ following $W_i(O)$ will read O_i

Snapshot Isolation: Example

T_1	T_2	T_3	**Comments**
$R_1(B)$			B_0
	$R_2(A)$		A_0
$W_1(B)$			B_1
$Commit_1$			
	$R_2(B)$		B_0
	$W_2(A)$		A_2
		$R_3(A)$	A_0
		$R_3(B)$	B_1
		$Commit_3$	
	$Commit_2$		

SI: First Committer Wins Rule

- If two concurrent Xacts update the same object, only one of them can commit
- First Committer Wins (FCW) Rule:
  - Before commiting a Xact T, the system checks if there exists a committed concurrent Xact T' that has updated some object that T intends to update
  - If T' exists, then T aborts
  - Otherwise, T commits with its updates written to the database
- **Example 1**:

T_1:	$R_1(X)$		$W_1(X)$			$\overset{Abort_1}{\cancel{Commit_1}}$
T_2:		$R_2(X)$		$W_2(X)$	$Commit_2$	

- **Example 2**:

T_1:	$R_1(X)$		$W_1(X)$		$Commit_1$	
T_2:		$R_2(X)$		$W_2(X)$		$\overset{Abort_2}{\cancel{Commit_2}}$

SI: First Updater Wins Rule

- Some implementations use write locks to prevent concurrent updates to same objects
- FCW rule becomes First Updater Wins (FUW) Rule
- First Updater Wins Rule:
  - Before a Xact *T* can update an object *O*, *T* needs to request for a write lock on *O*
  - If another Xact is holding a write lock on *O*, *T* is blocked
  - When *T*'s lock request on *O* is granted, the system checks if some other committed concurrent Xact has updated *O*
  - If so, *T* aborts
  - Otherwise, *T* updates *O* and proceeds with its execution

Snapshot Isolation Tradeoffs

- Performance often similar to Read Committed
- Does not suffer from lost update or unrepeatable read anomalies
- But vulnerable to some non-serializable executions
  - Write Skew Anomaly
  - Read-Only Transaction Anomaly

SI: Write Skew Anomaly

T_1	T_2	**Comments**
$R_1(A)$		A_0
	$R_2(A)$	A_0
$R_1(B)$		B_0
	$R_2(B)$	B_0
$W_1(A)$		A_1
$Commit_1$		
	$W_2(B)$	B_2
	$Commit_2$	

- The above schedule is a SI schedule but it is not conflict serializable

SI: Read-Only Transaction Anomaly

T_1	T_2	T_3	**Comments**
$R_1(B)$			B_0
	$R_2(A)$		A_0
$W_1(B)$			B_1
$Commit_1$			
	$R_2(B)$		B_0
	$W_2(A)$		A_2
		$R_3(A)$	A_0
		$R_3(B)$	B_1
		$Commit_3$	
	$Commit_2$		

- The above schedule is a SI schedule but it is not conflict serializable

Snapshot Isolation: Challenges

- Consider a set of transactional programs $\{P_1, \cdots, P_k\}$ in an application A
- A is a serializable application if every schedule arising out of executions of the programs of A is a serializable schedule

Given an application A,

1. Is A serializable under SI?
2. If not, how to make A serializable under SI?

Data Versions

- Let x_i be the version of data item x produced by T_i
- Let x_j be the version of data item x produced by T_j
- x_j is the immediate successor of x_i if
    1. T_i commits before T_j, and
    2. no transaction that commits between T_i's and T_j's commits produces a version of x

Transactional Dependencies

- ww dependency from T_1 to T_2
  - T_1 writes a version of some data item x, and
  - T_2 later writes the immediate successor version of x
- wr dependency from T_1 to T_2
  - T_1 writes a version of some data item x, and
  - T_2 reads this version of x
- rw dependency from T_1 to T_2
  - T_1 reads a version or some data item x, and
  - T_2 later creates the immediate successor version of x
- The above definitions assume data item reads/writes & can be generalized for predicate reads/writes

Dependency Serialization Graph (DSG)

- Consider a schedule S consisting of a set of committed transactions $\{T_1, \cdots, T_k\}$
- DSG(S) is an edge-labelled directed graph (V, E)
- V represents transactions $\{T_1, \cdots, T_k\}$
- E represents transactional dependencies
  - $T_i \stackrel{ww}{\rightarrow} T_j$
  - $T_i \stackrel{wr}{\rightarrow} T_j$
  - $T_i \stackrel{rw}{\rightarrow} T_j$
- Edge types:
  - $\dashrightarrow$ if transaction pair is concurrent
  - $\longrightarrow$ if transaction pair is non-concurrent

DSG: Example

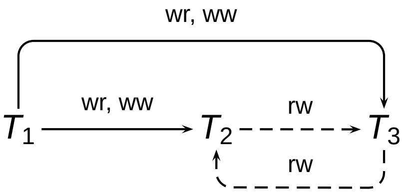
Schedule S:

$W_1(x)$, $W_1(y)$, $W_1(z)$, C_1,

$R_2(x)$, $W_2(y)$, C_2,

$W_3(x)$, $R_3(y)$, C_3

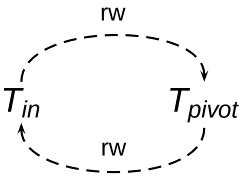
DSG(S):



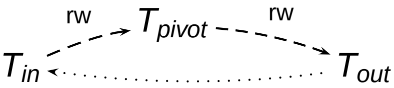
Non-Serializable SI schedules

Theorem 1: If S is a non-serializable SI schedule, then

1. There is at least one cycle in $DSG(S)$, and
2. For each cycle in $DSG(S)$, there exists three transactions, T_{in}, T_{pivot}, and T_{out} such that
   - T_{in} & T_{out} are possibly the same transaction,
   - T_{in} & T_{pivot} are concurrent with an edge $T_{in} \xrightarrow{rw} T_{pivot}$, and
   - T_{pivot} & T_{out} are concurrent with an edge $T_{pivot} \xrightarrow{rw} T_{out}$.



Refer to T_{pivot} as pivot transaction



Example 1: Write Skew Anomaly

Schedule S:

$R_1(a)$, $R_1(b)$, $W_1(a)$, C_1,

$R_2(a)$, $R_2(b)$, $W_2(b)$, C_2

DSG(S):

$$\begin{tikzcd} T_1 \arrow[r, bend left, dashed, "\text{rw}"] & T_2 \arrow[l, bend left, dashed, "\text{rw}"] \end{tikzcd}$$

Example 2: Read-only Xact Anomaly

Schedule S:

$R_1(b)$, $W_1(b)$, C_1,

$R_2(a)$, $R_2(b)$, $W_2(a)$, C_2

$R_3(a)$, $R_3(b)$, C_3,

DSG(S):

$$\begin{tikzcd} T_3 \arrow[r, dashed, "\mathrm{rw}"] & T_2 \arrow[r, dashed, "\mathrm{rw}"] & T_1 \arrow[ll, bend left, "\mathrm{wr}"] \end{tikzcd}$$

Application Programs & Transactions

Program P

S_1;
if (C) then
  S_2;
  S_3;
else
  S_4;
S_5;

Transaction T_1

if not(C) then abort; S_1; S_2; S_3; S_5

Transaction T_2

if (C) then abort; S_1; S_4; S_5

Static Dependencies

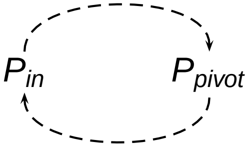
- Consider a set of transactional programs $\{P_1, \cdots, P_k\}$ in an application A
- There is a static α dependency from program P_i to program P_j, denoted by $P_i \xrightarrow{\alpha} P_j$, if there is a SI schedule S containing transactions T_i & T_j such that
  - program P_i gives rise to T_i,
  - program P_j gives rise to T_j, and
  - $T_i \xrightarrow{\alpha} T_j$, where $\alpha \in \{ww, rw, wr\}$.
- Furthermore, the static dependency is vulnerable, denoted by $P_i \dashrightarrow P_j$, if T_i and T_j are concurrent in S
  - $P_i \dashrightarrow P_j$ implies $T_i \xrightarrow{rw} T_j$

Static Dependency Graph (SDG)

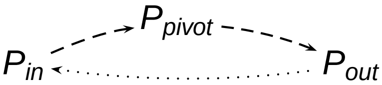
- Consider a set of transactional programs $\{P_1, \cdots, P_k\}$ in an application A
- SDG(A) is a directed graph (V, E)
- V represents programs $\{P_1, \cdots, P_k\}$
- E represents static dependencies among programs
- Non-vulnerable dependencies
  - $P_i \stackrel{ww}{\rightarrow} P_j$
  - $P_i \stackrel{wr}{\rightarrow} P_j$
  - $P_i \stackrel{rw}{\rightarrow} P_j$
- Vulnerable dependencies
  - $P_i \dashrightarrow P_j$

Dangerous Structures

- A SDG graph G contains a dangerous structure if there exists three nodes P_{in}, P_{pivot}, and P_{out} in G such that
  1. $P_{in} \dashrightarrow P_{pivot}$,
  2. $P_{pivot} \dashrightarrow P_{out}$, and
  3. either $P_{in} = P_{out}$ or G contains a path from P_{out} to P_{in}



Refer to P_{pivot} as pivot program



Serializable SI applications

Theorem 2: If an application A has a static dependency graph $SDG(A)$ with no dangerous structure, then A is serializable under SI

How to avoid SI anomalies

- Approach 1: Use a mix of SI and S2PL isolation levels
- Modify programs to eliminate dangerous structures in SDG(A)
  - Approach 2: Materialization
  - Approach 3: Promotion

Approach 1: Use both SI & S2PL

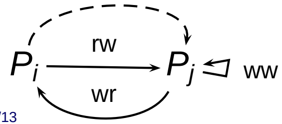
- DBMSs that support both S2PL & SI:
  - Microsoft SQL Server
  - MySQL
- Run the **pivot programs** using S2PL instead of SI
- Implementation details:
  - For each object version O_i created by a transaction T_i under SI,
    - ⋆ O_i must be protected by an exclusive lock until T_i commits

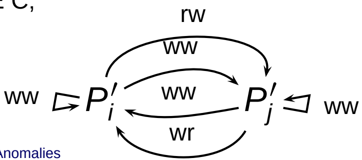
Program Modification Techniques

- Let A be an application with some dangerous structure in SDG(A)
- For each **dangerous structure** in SDG(A),
  - Choose one of its **vulnerable edges** (say (P_i, P_j))
  - Eliminate this vulnerable edge by modifying one or both of P_i and P_j
- Make (P_i, P_j) non-vulnerable by forcing the Xact pair execution to become non-concurrent
  - Introduce an extra ww dependency between P_i & P_j
- Modifications to a program must not change its functionality!
- Choosing a minimal set of vulnerable edges to modify to make A serializable under SI is NP-hard

Approach 2: Materialization technique

- Let (P_i, P_j) be a vulnerable edge to be modified
  - **P$_\mathbf{i}$**: SELECT ... FROM R WHERE C;
  - **P$_\mathbf{j}$**: UPDATE R SET ... WHERE C;
- Create a table `Conflict` (<u>id</u>, val)
- Insert a row $(id_{i,j}, 0)$ into `Conflict` to represent the vulnerable edge (P_i, P_j)
  - $id_{i,j}$ is a unique id for (P_i, P_j)
- Add an `UPDATE` statement to each of P_i & P_j:
  - **P$'_\mathbf{i}$**: UPDATE Conflict SET val=val+1 WHERE id=$id_{i,j}$;
    SELECT ... FROM R WHERE C;
  - **P$'_\mathbf{j}$**: UPDATE Conflict SET val=val+1 WHERE id=$id_{i,j}$;
    UPDATE R SET ... WHERE C;





Approach 3: Promotion technique

- Let (P_i, P_j) be a vulnerable edge to be modified
  - $\mathbf{P_i}$: SELECT ... FROM R WHERE C;
  - $\mathbf{P_j}$: UPDATE R SET ... WHERE C;
- Promote the rw dependency to a ww dependency by adding an **identity write** to P_i:
  - $\mathbf{P_i'}$: UPDATE R SET col = col WHERE C;
    SELECT ... FROM R WHERE C;
- The promotion can also be achieved by using `FOR UPDATE` clause in `SELECT` statement:
  - $\mathbf{P_i'}$: SELECT ... FROM R WHERE C FOR UPDATE;

Approach 3: Promotion technique (cont.)

- Caveat: Not applicable if the rw dependency involves a read that is part of a selection predicate as it could be vulnerable to phantom problem

Serializable Snapshot Isolation (SSI)

- Idea developed around 2008
- Implemented in PostgreSQL 9.1 in 2011

Quiz

Draw the SDG graph for the following banking application (with five transaction programs) and discuss the ways to make the application serializable under Snapshot Isolation.

Database Schema:

- Account (Name, CustomerID)
- Saving (CustomerID, Balance)
- Checking (CustomerID, Balance)

- The Account table represents the customers with a non-null and unique constraint on Account.CustomerID.
- Checking.Balance and Savings.Balance are numeric valued, each representing the balance in the corresponding account for one customer.

Quiz (cont.)

Transaction Programs

1. Balance(N) is a parameterized transaction that represents calculating the total balance for a customer. It looks up Account to get the CustomerID value for N, and then returns the sum of savings and checking balances for that CustomerID.

2. DepositChecking(N,V) is a parameterized transaction that represents making a deposit on the checking account of a customer. Its operation is to look up the Account table to get CustomerID corresponding to the name N and increase the checking balance by V for that CustomerID.

3. TransactSaving(N, V) represents making a deposit or withdrawal on the savings account. It increases the savings balance by V for that customer.

Quiz (cont.)

4. Amalgamate(N1, N2) represents moving all the funds from one customer to another. It reads the balances for both accounts of customer N1, then sets both to zero, and finally increases the checking balance for N2 by the sum of N1's previous balances.

5. WriteCheck(N,V) represents writing a check against an account. Its operation is to look up Account to get the CustomerID value for N, evaluate the sum of savings and checking balances for that CustomerID. If the sum is less than V, it decreases the checking balance by $V + 1$ (reflecting a penalty of 1 for overdrawing), otherwise it decreases the checking balance by V.

Balance(N) transaction

```
SELECT custid INTO cid
FROM Account
WHERE name=n;

SELECT bal INTO a
FROM Saving
WHERE custid=cid;

SELECT bal INTO b
FROM Checking
WHERE custid=cid;

total:=a+b;
```

DepositCecking(N,V) transaction

```
SELECT CustomerId INTO :x
FROM Account
WHERE Name=:N;

SELECT Balance INTO :b
FROM Checking
WHERE CustomerId=:x;

UPDATE Checking
SET Balance = Balance+:V
WHERE CustomerId=:x;
```

TransactSaving(N,V) transaction

```
SELECT CustomerId INTO :x
FROM Account
WHERE Name=:N;

SELECT Balance INTO :a
FROM Saving
WHERE CustomerId=:x;

UPDATE Saving
SET Balance = Balance+:V
WHERE CustomerId=:x;
```

Amalgamate(N1,N2) transaction

```
SELECT CustomerId INTO :x
FROM Account
WHERE Name=:N1;

SELECT CustomerId INTO :y
FROM Account
WHERE Name=:N2;

SELECT Balance INTO :a
FROM Saving
WHERE CustomerId=:x;
```

Amalgamate(N1,N2) transaction (cont.)

```
SELECT Balance INTO :b
FROM Checking
WHERE CustomerId=:x;

Total := :a+:b;

UPDATE Saving
SET Balance = 0.0
WHERE CustomerId=:x;

UPDATE Checking
SET Balance = 0.0
WHERE CustomerId=:x;
```

Amalgamate(N1,N2) transaction (cont.)

```
UPDATE Checking
SET Balance = Balance + :Total
WHERE CustomerId=:y;
```

WriteCheck(N,V) transaction

```
SELECT CustomerId INTO :x
FROM Account
WHERE Name=:N;

SELECT Balance INTO :a
FROM Saving
WHERE CustomerId=:x;

SELECT Balance INTO :b
FROM Checking
WHERE CustomerId=:x;
```

WriteCheck(N,V) transaction (cont.)

```
IF (:a+:b) < :V THEN
     UPDATE Checking
     SET Balance = Balance-(:V+1)
     WHERE CustomerId=:x;
ELSE
     UPDATE Checking
     SET Balance = Balance-:V
     WHERE CustomerId=:x;
END IF;
```

References

Required Readings

- M. Alomari, M. Cahill, A. Fekete, U. Roehm, *The cost of serializability on platforms that use snapshot isolation*, ICDE 2008.
- A. Fekete, D. Liarokapis, P. O'Neil, E. O'Neil, D. Shasha, *Making snapshot isolation serializable*, ACM TODS, 30(2), 492-528, 2005.

Additional Readings

- A. Fekete, *Allocating isolation levels to transactions*, PODS 2005.