CS5226 Lecture 6
Transaction Tuning

Transaction

- Transaction (Xact) - any one execution of a user program
  - an abstraction representing a logical unit of work.
- A Xact T_i can be viewed as a sequence of actions:
  - $R_i(O)$ = T_i reads an object O
  - $W_i(O)$ = T_i writes an object O
  - $Commit_i$ = T_i completes successfully
  - $Abort_i$ = T_i terminates unsuccessfully and undoes all previous actions
- Each Xact must end with either a commit or an abort action

Transaction (cont.)

- **Example**: A Xact T_1 that transfers \$100 from A to B can be abstracted as

  $$T_1\text{: } R_1(\text{A}), W_1(\text{A}), R_1(\text{B}), W_1(\text{B}), Commit_1$$

- Xact consistency property: If a Xact starts with a consistent DB state, then it also ends with a consistent DB state
- But, an intermediate DB state is not necessarily consistent!

Transaction Schedule

- Schedule = a list of actions from a set of Xacts, where the order of the actions within each Xact is preserved
- **Example**: Consider two Xacts T_1 and T_2:
  - T_1 transfers \$100 from A to B

    T_1: $R_1(A)$, $W_1(A)$, $R_1(B)$, $W_1(B)$, $Commit_1$
  - T_2 increments both A and B by 10%

    T_2: $R_2(A)$, $W_2(A)$, $R_2(B)$, $W_2(B)$, $Commit_2$
- Some schedules of T_1 and T_2:

  S_1: $R_1(A),W_1(A),R_1(B),W_1(B),Commit_1, R_2(A),W_2(A),R_2(B),W_2(B),Commit_2$

  S_2: $R_2(A),W_2(A),R_2(B),W_2(B),Commit_2, R_1(A),W_1(A),R_1(B),W_1(B),Commit_1$

  S_3: $R_1(A),W_1(A),R_2(A),W_2(A),R_1(B), W_1(B),Commit_1,R_2(B),W_2(B),Commit_2$
- A serial schedule is a schedule where the actions of Xacts are not interleaved

Read/Write From & Final Write

- We say that T_j reads O from T_i in a schedule S if the last write action on O before $R_j(O)$ in S is $W_i(O)$
- We say that T_i performs the final write on O in a schedule S if the last write action on O in S is $W_i(O)$
- **Example**: Consider the following schedule.

$R_1(A), W_1(A), R_2(A), W_2(A), R_1(B), W_1(B), Commit_1, R_2(B), W_2(B), Commit_2$

- T_2 reads A from T_1
- T_2 reads B from T_1
- T_2 performs the final write on A
- T_2 performs the final write on B

Interleaved Transaction Execution

- **Example**: Consider the Xacts T_1 & T_2 and four possible schedules:

  T_1: transfers \$100 from A to B
  T_2: increments both A and B by 10%

  S_1: R_1(A),W_1(A),R_1(B),W_1(B),$Commit_1$, R_2(A),W_2(A),R_2(B),W_2(B),$Commit_2$
  S_2: R_2(A),W_2(A),R_2(B),W_2(B),$Commit_2$, R_1(A),W_1(A),R_1(B),W_1(B),$Commit_1$
  S_3: R_1(A),W_1(A),R_2(A),W_2(A),R_1(B), W_1(B),$Commit_1$,R_2(B),W_2(B),$Commit_2$
  S_4: R_1(A),W_1(A), R_2(A),W_2(A),R_2(B),W_2(B),$Commit_2$, R_1(B), W_1(B),$Commit_1$

If $A = \$300$ and $B = \$500$ then

- S_1 results in $A = \$220$ and $B = \$660$
- S_2 results in $A = \$230$ and $B = \$650$
- S_3 results in $A = \$220$ and $B = \$660$
- S_4 results in $A = \$220$ and $B = \$650$

Anomalies with Interleaved Executions

- Two actions on the same object conflict if (1) at least one of them is a write action and (2) the actions are from different Xacts
- Anomalies can arise due to conflicting actions:
  1. Dirty read problem (due to WR conflicts)
     - T_2 reads an object that has been modified by T_1 (which has not yet committed)
     - T_2 could see an inconsistent DB state!
  2. Unrepeatable read problem (due to RW conflicts)
     - T_2 updates an object that T_1 has just read while T_1 is still in progress
     - T_1 could get a different value if it reads the object again!
  3. Lost update problem (due to WW conflicts)
     - T_2 overwrites the value of an object that has been modified by T_1 while T_1 is still in progress
     - T_1's update is lost!

Dirty Read Problem: Example

T_1	T_2	**Comments**
		$x = 100$
R(x)		100
$x = x + 20$		
W(x)		$x = 120$
	R(x)	120
	$x = x \times 2$	
Rollback		
	W(x)	$x = 240$

- For every serial schedule, the final value of x is 200

Unrepeatable Read Problem: Example

T_1	T_2	**Comments**
		$x = 100$
R(x)		100
	R(x)	100
	$x = x - 20$	
	W(x)	$x = 80$
R(x)		80

- For every serial schedule, both values read by T_1 are the same

Lost Update Problem: Example

T_1	T_2	**Comments**
		$x = 100$
R(x)		100
	R(x)	100
$x = x + 20$		
	$x = x \times 2$	
W(x)		$x = 120$
	W(x)	$x = 200$

- For schedule (T_1, T_2), the final value of x is 240
- For schedule (T_2, T_1), the final value of x is 220

Conflict Serializable Schedule

- Two schedules S and S' are said to be conflict equivalent if
  1. they involve the same set of actions of the same Xacts, and
  2. they order every pair of conflicting actions of two committed Xacts in the same way
- **Example**: Consider the two schedules S and S':

$$S: \quad \cdots\cdots W_i(X) \cdots\cdots R_j(X) \cdots\cdots$$
$$S': \quad \cdots\cdots R_j(X) \cdots\cdots W_i(X) \cdots\cdots$$

  S is not conflict equivalent to S'
- A schedule is a conflict serializable schedule if it is conflict equivalent to a serial schedule

Conflict Serializable Schedule: Example

- **Example**: Consider the following transactions and schedule:

  T_1: $R_1(A)$, $W_1(A)$, $Commit_1$,
  T_2: $W_2(A)$, $Commit_2$
  T_3: $W_3(A)$, $Commit_3$

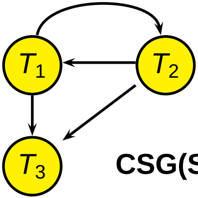
  S: $R_1(A)$, $W_2(A)$, $Commit_2$, $W_1(A)$, $Commit_1$, $W_3(A)$, $Commit_3$

- S is not a conflict serializable schedule
  - In any serial schedule S' with T_1 preceding T_2, the orderings of $W_1(A)$ and $W_2(A)$ are different in S & S'
  - In any serial schedule S' with T_2 preceding T_1, the orderings of $R_1(A)$ and $W_2(A)$ are different in S & S'

Testing for Conflict Serializability

- A conflict serializability graph for a schedule S is a directed graph $G = (V, E)$ such that
  - V contains a node for each committed Xact in S
  - E contains (T_i, T_j) if an action in T_i precedes and conflicts with one of T_j's actions
- **Theorem**: A schedule is conflict serializable iff its conflict serializability graph is acyclic
- **Example**: Conflict serializability graph for schedule

R_1(A), W_2(A), $Commit_2$, W_1(A), $Commit_1$, W_3(A), $Commit_3$



CSG(S)

Transaction Scheduler

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

Transaction Scheduler

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

- For each input action (read, write, commit, rollback) to the scheduler, the scheduler performs one of the following:
  - output the action to the schedule,
  - reject the action and abort the transaction, or
  - postpone the action by blocking the transaction

Lock-Based Concurrency Control

- Each Xact needs to request for an appropriate lock on an object before the Xact can access the object
- Locking modes
  - Shared (S) locks for reading objects
  - Exclusive (X) locks for writing objects
- Lock compatibility:

Lock Requested	Lock Held		
	-	S	X
S	√	√	×
X	√	×	×

√: Compatible
Lock request is granted

×: Incompatible
Lock request is blocked

- A Xact is blocked if its request for a lock on an object is incompatible with an existing lock on the object held by some other Xact.

Lock-Based Concurrency Control (cont.)

- Notations
  - $S_i(O)$: Xact T_i is requesting S-lock on object O
  - $X_i(O)$: Xact T_i is requesting X-lock on object O
  - $U_i(O)$: Xact T_i releases lock on object O
- **Example**: $R_1(A)$, $W_2(A)$, $W_2(B)$, $W_1(B)$

$S_1(A)$,$R_1(A)$,$U_1(A)$,$X_2(A)$, $W_2(A)$, $X_2(B)$, $W_2(B)$, $U_2(A)$,$U_2(B)$,$X_1(B)$, $W_1(B)$, $U_1(B)$

Two Phase Locking (2PL) Protocol

- 2PL Protocol:
    1. To read an object O, a Xact must hold a S-lock or X-lock on O
    2. To write to an object O, a Xact must hold a X-lock on O
    3. Once a Xact releases a lock, the Xact can't request any more locks
- Xacts using 2PL can be characterized into two phases:
    - Growing phase: before releasing 1^{st} lock
    - Shrinking phase: after releasing 1^{st} lock

Theorem: 2PL schedules are conflict serializable

Two Phase Locking (2PL) Protocol (cont.)

- **Example**: $R_1(A)$, $W_2(A)$, $W_2(B)$, $W_1(B)$

$S_1(A)$, $R_1(A)$, $\boxed{U_1(A)}$, $X_2(A)$, $W_2(A)$, $X_2(B)$, $W_2(B)$, $U_2(A)$, $U_2(B)$, $\boxed{X_1(B)}$, $W_1(B)$, $U_1(B)$

Not permitted by 2PL!

- The above example schedule is not a 2PL schedule

Strict Two Phase Locking (strict 2PL) Protocol

- Strict 2PL Protocol:
  1. To read an object O, a Xact must hold a S-lock or X-lock on O
  2. To write to an object O, a Xact must hold a X-lock on O
  3. A Xact must hold on to locks until Xact commits or aborts
- **Theorem**: strict 2PL schedules $\subset$ 2PL schedules

Deadlocks

- Deadlock: cycle of Xacts waiting for locks to be released by each other
- **Example**:
  T_1 requests X-lock on A and is granted;
  T_2 requests X-lock on B and is granted;
  T_1 requests X-lock on B and is blocked;
  T_2 requests X-lock on A and is blocked;

Phantom Read Phenomenon

Accounts

account	branch	balance
101	north	200
250	south	1000
444	south	800

Assets

branch	total
north	200
south	1800

Transaction 1	Transaction 2
select **sum** (balance) **from** Accounts **where** branch = 'south'; **select** total **from** Assets **where** branch = 'south';	**insert into** Accounts **values** (222,'south',100); **update** Assets **set** total = total + 100 **where** branch = 'south';

Phantom Read Phenomenon (cont.)

- A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that the set of rows satisfying the condition has changed due to another recently committed transaction.

```
select   sum (balance)
from     Accounts
where    branch = 'south';
                                insert into Accounts
                                   values (222,'south',100);

                                update Assets
                                   set total = total + 100
                                   where branch = 'south';
select   total
from     Assets
where    branch = 'south';
```

- Can be prevented by predicate locking via index locking

ANSI SQL Isolation Levels

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Read
READ UNCOMMITTED	possible	possible	possible
READ COMMITTED	not possible	possible	possible
REPEATABLE READ	not possible	not possible	possible
SERIALIZABLE	not possible	not possible	not possible

Degree	Isolation level	Write Locks	Read Locks	Phantom Read
0	Read Uncommitted	long duration	none	possible
1	Read Committed	long duration	short duration	possible
2	Repeatable Read	long duration	long duration	possible
3	Serializable	long duration	long duration	not possible

Transaction Chopping

Transaction T_1

BEGIN TRANSACTION;
SQL statements$_1$;
SQL statements$_2$;
SQL statements$_3$;
END TRANSACTION;

Transaction $T_{1,1}$

BEGIN TRANSACTION;
SQL statements$_1$;
END TRANSACTION;

Transaction $T_{1,2}$

BEGIN TRANSACTION;
SQL statements$_2$;
END TRANSACTION;

Transaction $T_{1,3}$

BEGIN TRANSACTION;
SQL statements$_3$;
END TRANSACTION;

Transaction Chopping (cont.)

- A chopping partitions T into k pieces, $k \geq 1$
- Each database access performed by T is in exactly one piece
- **Example 1**: Consider 2 transactions:
  - T_1: $R_1(a), W_1(a), R_1(y), W_1(y)$
  - T_2: $R_2(a)$
- T_1 could be chopped into 2 pieces:
  - $T_{1,1}$: $R_1(a), W_1(a)$ $T_{1,2}$: $R_1(y), W_1(y)$

Why Transaction Chopping?

- Transaction chopping can increase concurrency

- Possible S2PL schedules for $\{T_1, T_2\}$:

  S1. $R_2(a), R_1(a), W_1(a), R_1(y), W_1(y)$
  S2. $R_1(a), R_2(a), W_1(a), R_1(y), W_1(y)$
  S3. $R_1(a), W_1(a), R_1(y), W_1(y), R_2(a)$

- Possible S2PL schedules for $\{T_{1,1}, T_{1,2}, T_2\}$

  S1. $R_2(a), R_1(a), W_1(a), R_1(y), W_1(y)$
  S2. $R_1(a), R_2(a), W_1(a), R_1(y), W_1(y)$
  S3. $R_1(a), W_1(a), R_1(y), W_1(y), R_2(a)$
  S4. $R_1(a), W_1(a), R_2(a), R_1(y), W_1(y)$
  S5. $R_1(a), W_1(a), R_1(y), R_2(a), W_1(y)$

  Each of the 5 schedules is a conflict serializable schedule for $\{T_1, T_2\}$

CSG(S,T)

- Let T denote a set of transactions
- Let T' denote a chopping of T
- Let S denote a schedule for T'
- We use $CSG(S, T')$ to denote the conflict serializability graph for schedule S wrt the set of transactions T'

- If S is a S2PL schedule for T', then
  - $CSG(S, T')$ must be acyclic
  - But $CSG(S, T)$ could be cyclic

Incorrect Transaction Chopping

- **Example 2**: Consider 3 transactions:
  - T_1: $R_1(a)$, $W_1(a)$, $R_1(y)$, $W_1(y)$
  - T_2: $R_2(b)$, $W_2(b)$, $R_2(y)$, $W_2(y)$
  - T_3: $R_3(a)$, $R_3(b)$, $R_3(y)$
- Suppose each of T_1 & T_2 is chopped into 2 pieces:
  - $T_{1,1}$: $R_1(a)$, $W_1(a)$ $T_{1,2}$: $R_1(y)$, $W_1(y)$
  - $T_{2,1}$: $R_2(b)$, $W_2(b)$ $T_{2,2}$: $R_2(y)$, $W_2(y)$
- A possible S2PL schedule for $\{T_{1,1}, T_{1,2}, T_{2,1}, T_{2,2}, T_3\}$:

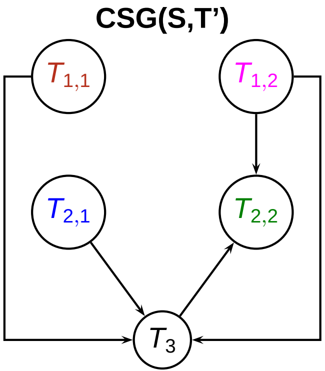
$R_1(a)$, $W_1(a)$, $R_1(y)$, $W_1(y)$, $R_2(b)$, $W_2(b)$, $R_3(a)$, $R_3(b)$, $R_3(y)$, $R_2(y)$, $W_2(y)$

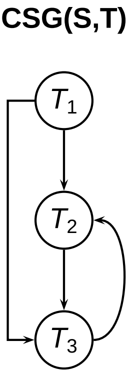
Not a conflict serializable schedule for $\{T_1, T_2, T_3\}$!

Incorrect Transaction Chopping (cont.)

- $T = \{T_1, T_2, T_3\}$
- $T' = \{T_{1,1}, T_{1,2}, T_{2,1}, T_{2,2}, T_3\}$
- A S2PL schedule S for T':

$R_1(a), W_1(a), R_1(y), W_1(y), R_2(b), W_2(b), R_3(a), R_3(b), R_3(y), R_2(y), W_2(y)$





Rollback-Safe Chopping

Transaction T_1

```
BEGIN TRANSACTION;
SQL statements₁;
if (condition₁) then
    ROLLBACK;
SQL statements₂;
if (condition₂) then
    ROLLBACK;
SQL statements₃;
END TRANSACTION;
```

Transaction $T_{1,1}$

```
BEGIN TRANSACTION;
SQL statements₁;
if (condition₁) then ROLLBACK;
END TRANSACTION;
```

Transaction $T_{1,2}$

```
BEGIN TRANSACTION;
SQL statements₂;
if (condition₂) then ROLLBACK;
END TRANSACTION;
```

Transaction $T_{1,3}$

```
BEGIN TRANSACTION;
SQL statements₃;
END TRANSACTION;
```

Rollback-Safe Chopping (cont.)

Transaction T_1

```
BEGIN TRANSACTION;
SQL statements1;
if (condition1) then
    ROLLBACK;
SQL statements2;
if (condition2) then
    ROLLBACK;
SQL statements3;
END TRANSACTION;
```

Transaction $T_{1,1}$

```
BEGIN TRANSACTION;
SQL statements1;
if (condition1) then ROLLBACK;
SQL statements2;
if (condition2) then ROLLBACK;
END TRANSACTION;
```

Transaction $T_{1,2}$

```
BEGIN TRANSACTION;
SQL statements3;
END TRANSACTION;
```

Rollback-Safe Chopping (cont.)

- A chopping of T is rollback-safe if
  - either T has no rollback statements
  - or all rollback statements of T are in its first piece
- All the statements in the first piece must execute before any other statement of T
- A chopping of a set of transactions is rollback-safe if the chopping of each of its transaction is rollback-safe

Execution rules

1. Each piece acts like a transaction: follows strict 2PL protocol
2. Order of execution of pieces follow order of transaction
3. If a piece is aborted because of a lock conflict, then it will be resubmitted repeatedly until it commits.
4. If a piece is aborted because of a rollback, then no other pieces for that transaction will execute.

Correct Chopping

- A chopping of $\{T_1, \cdots, T_n\}$ is correct if **any** execution of the chopping that obeys the execution rules is *conflict equivalent* to some serial execution of the original transactions.

Conflicting Pieces

Two pieces p and p' are conflicting if all the following conditions hold:

1. p & p' belong to different transactions,
2. p contains a step that writes an object O, and
3. p' contains a step that reads/writes the same object O

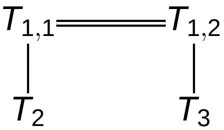
Chopping Graph

- Undirected graph
- Each vertex represents a chopped piece
- Two types of edges:
  - conflict edges (C-edges)
  - sibling edges (S-edges)
- (p, p') is a **S-edge** if p & p' belong to the same transaction
- (p, p') is a **C-edge** if p & p' is a pair of conflicting pieces

Example 3

- T_1: R_1(x), W_1(x), R_1(y), W_1(y)
- T_2: R_2(x), W_2(x)
- T_3: R_3(y), W_3(y)

- Suppose that T_1 is chopped into two pieces:
  - $T_{1,1}$: R_1(x), W_1(x)
  - $T_{1,2}$: R_1(y), W_1(y)

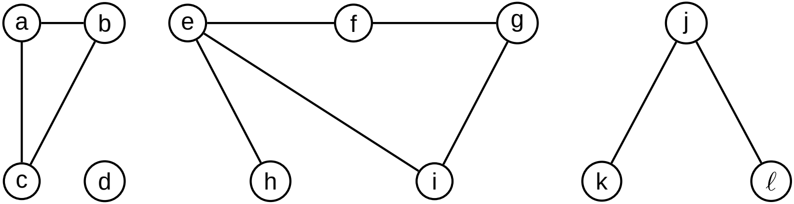


S-edge: =

C-edge: —

Review: Graph Concepts

- A simple cycle consists of a sequence of distinct nodes $< v_1, v_2, \cdots, v_n >$ such that
  1. there is an edge (v_i, v_{i+1}) for each $i \in [1, n)$, and
  2. there is an edge (v_n, v_1)
- A connected component C in a graph G is a maximal subgraph of G such that for every pair of nodes v & v' in C, v is connected to v' by some path



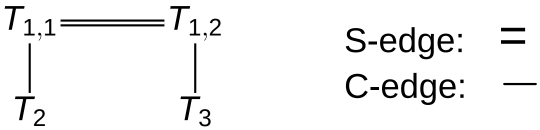
SC-cycle

- A chopping graph has a SC-cycle if it contains a simple cycle that includes at least one S-edge and at least one C-edge

Revisiting Example 3

- T_1: R_1(x), W_1(x), R_1(y), W_1(y)
- T_2: R_2(x), W_2(x)
- T_3: R_3(y), W_3(y)

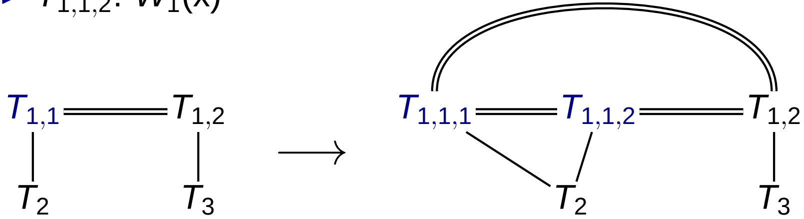
- Suppose that T_1 is chopped into two pieces:
  - $T_{1,1}$: R_1(x), W_1(x)
  - $T_{1,2}$: R_1(y), W_1(y)



Revisiting Example 3 (cont.)

Suppose that $T_{1,1}$ is further chopped into two pieces:

- $T_{1,1,1}$: $R_1(\mathrm{x})$
- $T_{1,1,2}$: $W_1(\mathrm{x})$



- $T_{1,2}$: $R_1(\mathrm{y})$, $W_1(\mathrm{y})$
- T_2: $R_2(\mathrm{x})$, $W_2(\mathrm{x})$
- T_3: $R_3(\mathrm{y})$, $W_3(\mathrm{y})$

Correct Chopping

Theorem 1: A chopping is correct if

1. it is rollback-safe, and
2. its chopping graph contains no SC-cycle

Quiz 1

- Consider the following transactions
  - T_1: R_1(x), W_1(x), R_1(y), W_1(y)
  - T_2: R_2(x)
  - T_3: R_3(y), W_3(y)
- Suppose that T_1 is chopped into 3 pieces:
  - $T_{1,1}$: R_1(x)
  - $T_{1,2}$: W_1(x)
  - $T_{1,3}$: R_1(y), W_1(y)
- Is this chopping correct?

Properties of Chopping

Lemma 1: If a chopping graph contains an SC-cycle, then any further chopping of any of the transactions will not render it acyclic

Lemma 2: If two pieces of transaction T are in an SC-cycle as the result of some chopping, then they will be in a cycle even if no other transactions are chopped.

Private chopping

- Consider a set of transactions $T = \{T_1, \cdots, T_n\}$ that run at an interval
- Let $chop(T_i)$ denote a chopping of T_i
- $chop(T_i)$ is a private chopping of T_i, denoted by private(T_i), if
  1. $chop(T_i)$ is a rollback-safe chopping of T_i, and
  2. there is no SC-cycle in the graph whose nodes are $(T - \{T_i\}) \cup chop(T_i)$

Theorem 2:
The chopping consisting of $\{private(T_1), \cdots, private(T_n)\}$ is rollback safe and has no SC-cycles.

Finest chopping

- Let FineChop(T_i) denote the finest chopping of T_i with the following two properties:
  1. FineChop(T_i) is a private chopping of T_i
  2. If piece p is a member of FineChop(T_i), then there is no other private chopping of T_i containing p_1 and p_2 such that p_1 and p_2 partition p and neither is empty

The finest chopping of a set of transactions $S = \{T_1, \cdots, T_n\}$ is given by $\{$ FineChop(T_1), $\cdots$, FineChop(T_n) $\}$

Finest chopping: algorithm

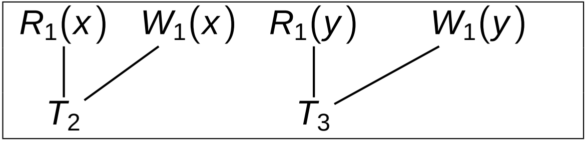
Input: A set of transactions S & a transaction $T \in S$
Output: FineChop(T)

01. if there are rollback statements in T then
02. $\quad$ $p_1 \leftarrow$ all database writes of T that may occur before or concurrently with any rollback statement of T
03. else
04. $\quad$ $p_1 \leftarrow$ set consisting of the first database access
05. $P \leftarrow \{\{x\} \mid x$ is a database access not in $p_1\}$
06. $P \leftarrow P \cup \{p_1\}$
07. let $G = (V, E)$ be an undirected graph, where $V = (S - \{T\}) \cup P$, E is the set of C-edges among V
08. for each connected component C_i in G do
09. $\quad$ merge all pieces of P in C_i into a single piece
10. return the set of pieces in P

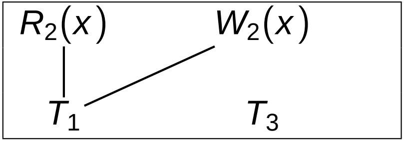
Revisiting Example 3

- T_1: R_1(x), W_1(x), R_1(y), W_1(y)
- T_2: R_2(x), W_2(x)
- T_3: R_3(y), W_3(y)

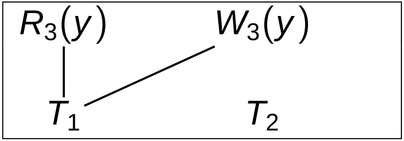
FineChop(T_1) = $\{\{R_1(x), W_1(x)\}, \{R_1(y), W_1(y)\}\}$



FineChop(T_2) = $\{\{R_2(x), W_2(x)\}\}$



FineChop(T_3) = $\{\{R_3(y), W_3(y)\}\}$



Quiz 2

Find the finest chopping for the set of 4 transactions:

- T_1: R_1(a), W_1(a), R_1(x), W_1(x)
- T_2: R_2(c), W_2(c), R_2(x), W_2(x)
- T_3: R_3(d), W_3(d), R_3(y), W_3(y)
- T_4: R_4(a), R_4(b), R_4(c), R_4(x), R_4(d), R_4(e), R_4(y)

References

Required Readings

- Section 2.2, Locking and Concurrency Control, Shasha & Bonnet's book
- Appendix B, Transaction Chopping, Shasha & Bonnet's book

Additional Readings

- D. Shasha, et al., Transaction Chopping: Algorithms and Performance Studies, ACM TODS 20(3), 325-363, 1995