TRANSACTION CONCEPT

Transaction Concept

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items
- For example, transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

Required Properties of a Transaction

Required Properties of a Transaction

- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database

Transaction to transfer $50 from account A to account B:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

Required Properties of a Transaction

Required Properties of a Transaction

- **Consistency requirement**
  - In example, the sum of A and B is unchanged by the execution of the transaction
  - In general, consistency requirements include
    - Explicitly specified integrity constraints
      - primary keys and foreign keys
    - Implicit integrity constraints
      - sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction, when starting to execute, must see a consistent database
  - During transaction execution the database may be temporarily inconsistent
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Required Properties of a Transaction (Cont.)

Required Properties of a Transaction (Cont.)

- **Isolation requirement**
  - If between steps 3 and 6 (of the fund transfer transaction) , another transaction **T2** is allowed to access the partially updated database, it will see an inconsistent database (the sum *A* + *B* will be less than it should be).

T1	T2
1. **read**(*A*)	
2. *A* := *A* – 50	
3. **write**(*A*)	
	read(A), read(B), print(A+B)
4. **read**(*B*)	
5. *B* := *B* + 50	
6. **write**(*B*	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other
- However, executing multiple transactions concurrently has significant benefits

Required Properties of a Transaction

- **Durability requirement**
  - Once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failure
  - Transaction to transfer $50 from account A to account B:
    1. **read**(A)
    2. $A := A – 50$
    3. **write**(A)
    4. **read**(B)
    5. $B := B + 50$
    6. **write**(B)

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

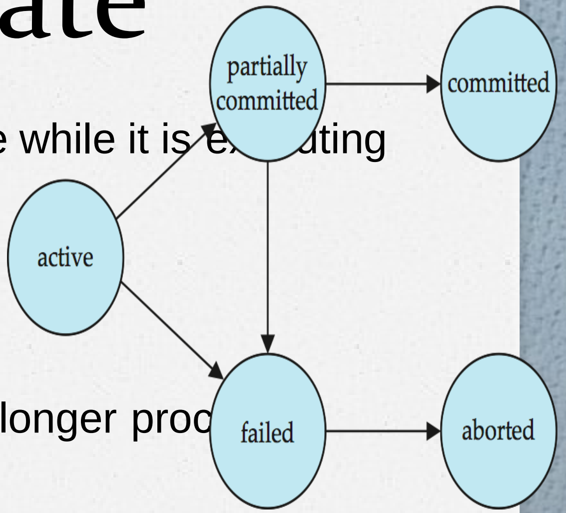
- **Atomicity**:
  - Either all operations of the transaction are properly reflected in the database or none are
- **Consistency**:
  - Execution of a transaction in isolation preserves the consistency of the database
- **Isolation**:
  - Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished
- **Durability**:
  - After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures

- Transaction Concept
- **Transaction State**
- Concurrent Executions

TRANSACTION STATE

Transaction State

Transaction State



- **Active**
  - The initial state; the transaction stays in this state while it is executing
- **Partially committed**
  - After the final statement has been executed
- **Failed**
  - After the discovery that normal execution can no longer proc
- **Aborted**
  - After the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
    - Restart the transaction
    - Kill the transaction

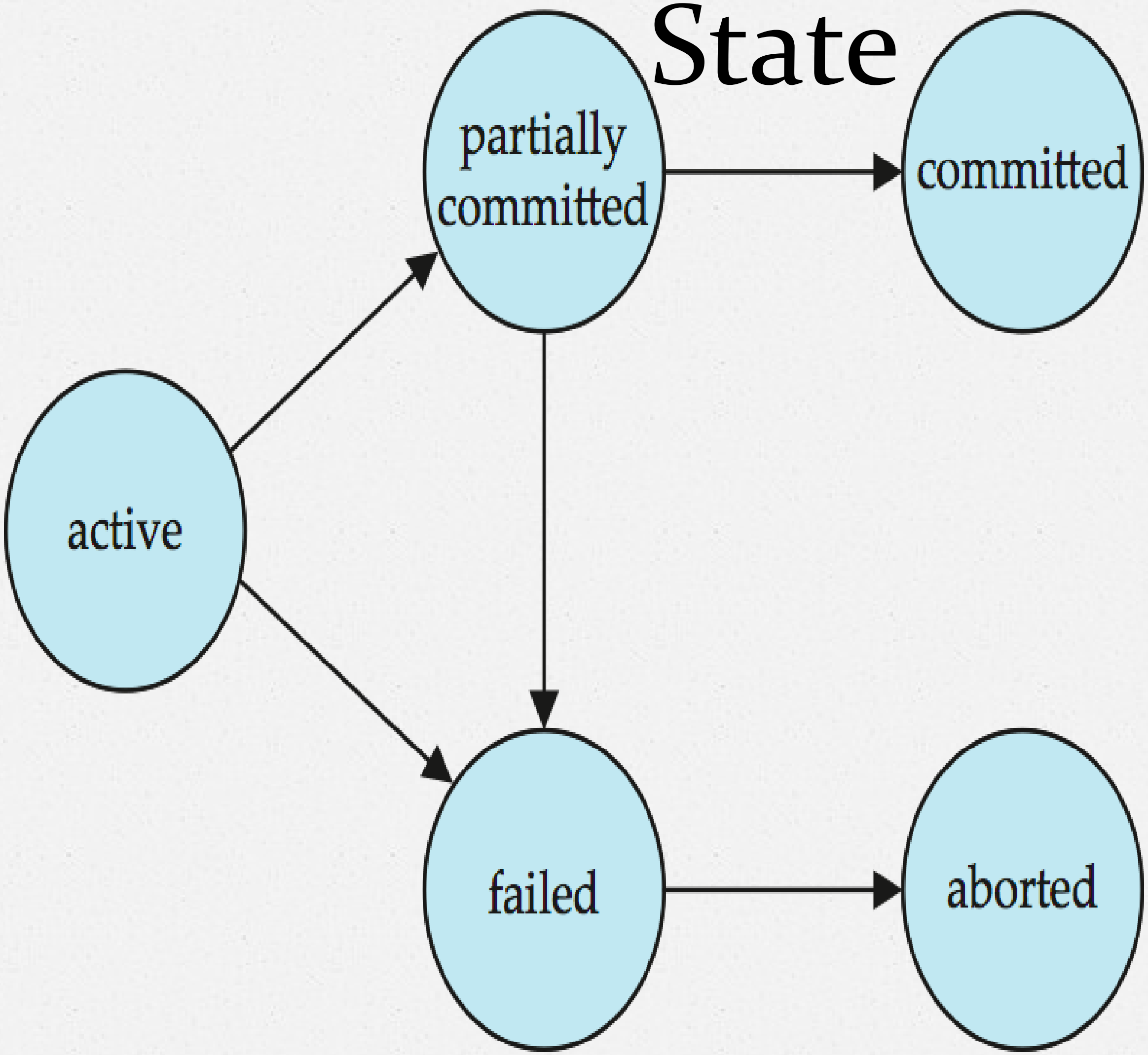
can be done only if no internal logical error

- **Committed**
  - After successful completion

Transaction to transfer \$50 from account A to account B:

1. **read**(*A*)
2. *A* := *A* – 50
3. **write**(*A*)
4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*)

Transitions for Transaction State



- Transaction Concept
- Transaction State
- **Concurrent Executions**

CONCURRENT EXECUTIONS

Concurrent Executions

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - For example, one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones

- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedul es

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a **commit** instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step.
- A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement

Schedule 1

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B
- An example of a **serial** schedule in which T_1 is followed by T_2 :

T_1	T_2
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
50	200	250	T1, write A	
50	250	300	T1, write B	@ Commit
45	250	295	T2, write	
45	255	30		Commit

Consistent @

Inconsistent @

Inconsistent @ Commit

Schedule 2

- A **serial** schedule in which T_2 is followed by T_1 :

T_1	T_2
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
commit	

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
90	200	290	T2, write A	
90	210	300	T2, write B	@ Commit
40	210	250	T1, write A	
40	260	300	T1, write B	@Commit

Consistent @

Inconsistent @

Inconsistent @ Commit

Values of A & B are different from Schedule 1 – yet consistent

Schedule 3

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is **equivalent** to Schedule 1

Schedule 3

T_1	T_2
read (A) $A := A - 50$ write (A)	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A)
read (B) $B := B + 50$ write (B) commit	
	read (B) $B := B + temp$ write (B) commit

Schedule 1

T_1	T_2
read (A) $A := A - 50$ write (A) read (B) $B := B + 50$ write (B) commit	
	read (A) $temp := A * 0.1$ $A := A - temp$ write (A) read (B) $B := B + temp$ write (B) commit

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
50	200	250	T1, write A	
45	200	245	T2, write A	
45	250	295	T1, write B	@ Commit
45	255	300	T2, write B	@Commit

Consistent @

Inconsistent @

Inconsistent @ Commit

Note – In schedules 1, 2 and 3, the sum "A + B" is preserved

Schedule 4

Schedule 4

- The following concurrent schedule does not preserve the sum of "*A* + *B*"

T_1	T_2
read (*A*) *A* := *A* – 50	
	read (*A*) *temp* := *A* * 0.1 *A* := *A* - *temp* write (*A*) read (*B*)
write (*A*) read (*B*) *B* := *B* + 50 write (*B*) commit	
	B := *B* + *temp* write (*B*) commit

A	B	A+B	Transaction	Remarks
100	200	300	@ Start	
90	200	290	T2, write A	
90	200	290	T1, write A	
90	250	340	T1, write B	@ Commit
90	260	350	T2, write B	@Commit

Consistent @

Inconsistent @

Inconsistent @ Commit

Module Summary

Module Summary

- A task is a database is done as a transaction that passes through several states
- Transactions are executed in concurrent fashion for better throughput
- Concurrent execution of transactions raise serializability issues that need to be addressed
- All schedules may not satisfy ACID properties

- **Recoverability and Isolation**
- Transaction Definition in SQL
- View Serializability

RECOVERABILITY AND ISOLATION

What is recovery?

What is recovery?

- Serializability helps to ensure Isolation and Consistency of a schedule
- Yet, the Atomicity and Consistency may be compromised in the face of system failures
- Consider a schedule comprising a single transaction (obviously serial):
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
  7. **commit** *// Make the changes permanent; show the results to the user*
- What if system fails after Step 3 and before Step 6?
  - Leads to inconsistent state
  - Need to rollback update of A
- This is known as **Recovery**

Recoverable Schedules

Recoverable Schedules

- **Recoverable schedule**
  - If a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i **must** appear before the commit operation of T_j.
- The following schedule is not recoverable if T_9 commits immediately after the read(A) operation

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

- If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable

Cascading Rollbacks

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

- If T_{10} fails, T_{11} and T_{12} must also be rolled back
- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

Cascadeless Schedules

- **Cascadeless schedules** — for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless
- Example of a schedule that is NOT cascadeless

T_{10}	T_{11}	T_{12}
read (A)		
read (B)		
write (A)		
	read (A)	
	write (A)	
		read (A)
abort		

Recoverable Schedules: Example

Recoverable Schedules: Example

- Irrecoverable Schedule

T1	T1's Buffer	T2	T2's Buffer	Database
				A = 5000
R(A);	A = 5000			A = 5000
A = A – 1000;	A = 4000			A = 5000
W(A);	A = 4000			A = 4000
		R(A);	A = 4000	A = 4000
		A = A + 500;	A = 4500	A = 4000
		W(A);	A = 4500	A = 4500
		Commit;		
Failure Point				
Commit;				

Recoverable Schedules: Example

Recoverable Schedules: Example

- Recoverable Schedule with cascading rollback

T1	T1's Buffer	T2	T2's Buffer	Database
				A = 5000
R(A);	A = 5000			A = 5000
A = A – 1000;	A = 4000			A = 5000
W(A);	A = 4000			A = 4000
		R(A);	A = 4000	A = 4000
		A = A + 500;	A = 4500	A = 4000
		W(A);	A = 4500	A = 4500
Failure Point				
Commit;				
		Commit;		

Recoverable Schedules: Example

Recoverable Schedules: Example

- Recoverable Schedule without cascading rollback

T1	T1's Buffer	T2	T2's Buffer	Database
				A = 5000
R(A);	A = 5000			A = 5000
A = A – 1000;	A = 4000			A = 5000
W(A);	A = 4000			A = 4000
Commit;				
		R(A);	A = 4000	A = 4000
		A = A + 500;	A = 4500	A = 4000
		W(A);	A = 4500	A = 4500
		Commit;		

- Recoverability and Isolation
- **Transaction Definition in SQL**
- View Serializability

TRANSACTION DEFINITION IN SQL

Transaction Definition in SQL

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction
- In SQL, a transaction begins implicitly
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one
  - **Rollback work** causes current transaction to abort

Transaction Control Language (TCL)

Transaction Control Language (TCL)

- The following commands are used to control transactions.
  - **COMMIT** − to save the changes
  - **ROLLBACK** − to roll back the changes
  - **SAVEPOINT** − creates points within the groups of transactions in which to ROLLBACK
  - **SET TRANSACTION** − Places a name on a transaction

- Transactional control commands are only used with the **DML Commands** such as
  - INSERT, UPDATE and DELETE only
  - They cannot be used while creating tables or dropping them because these operations are automatically committed in the database

TCL: COMMIT Command

TCL: COMMIT Command

- The COMMIT is the transactional command used to save changes invoked by a transaction to the database
- The COMMIT saves all the transactions to the database since the last COMMIT or ROLLBACK command
- The syntax for the COMMIT command is as follows:
  - SQL> DELETE FROM Customers WHERE AGE = 25;
  - SQL> COMMIT;

SQL> SELECT * FROM Customers;

Before DELETE

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
2	Khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

SQL> SELECT * FROM Customers;

After DELETE

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
3	kaushik	23	Kota	2000
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

TCL: ROLLBACK Command

TCL: ROLLBACK Command

- The ROLLBACK is the command used to undo transactions that have not already been saved to the database
- This can only be used to undo transactions since the last COMMIT or ROLLBACK command was issued
- The syntax for a ROLLBACK command is as follows:
  - SQL> DELETE FROM Customers WHERE AGE = 25;
  - SQL> ROLLBACK;

SQL> SELECT * FROM Customers;
* FROM Customers;

Before DELETE

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
2	Khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

SQL> SELECT

After DELETE

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
2	Khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

TCL: SAVEPOINT / ROLLBACK Command

TCL: SAVEPOINT / ROLLBACK Command

- A SAVEPOINT is a point in a transaction when you can roll the transaction back to a certain point without rolling back the entire transaction

- The syntax for a SAVEPOINT command is:
  - **SAVEPOINT SAVEPOINT_NAME;**

- This command serves only in the creation of a SAVEPOINT among all the transactional statements.
- The ROLLBACK command is used to undo a group of transactions
- The syntax for rolling back to a SAVEPOINT is:
  - **ROLLBACK TO SAVEPOINT_NAME;**

- Example:
  - SQL> SAVEPOINT SP1;
    - Savepoint created.
  - SQL> DELETE FROM Customers WHERE ID=1;
    - 1 row deleted.
  - SQL> SAVEPOINT SP2;
    - Savepoint created.
  - SQL> DELETE FROM Customers WHERE ID=2;
    - 1 row deleted.
  - SQL> SAVEPOINT SP3;
    - Savepoint created.
  - SQL> DELETE FROM Customers WHERE ID=3;
    - 1 row deleted.

TCL: SAVEPOINT / ROLLBACK Command

TCL: SAVEPOINT / ROLLBACK Command

- Three records deleted
- Undo the deletion of first two
- SQL> ROLLBACK TO SP2;
  - Rollback complete

```
SQL> SAVEPOINT SP1;
SQL> DELETE FROM Customers
WHERE ID=1;  SQL> SAVEPOINT
SP2;
SQL> DELETE FROM Customers
WHERE ID=2;  SQL> SAVEPOINT
SP3;
SQL> DELETE FROM Customers WHERE ID=3;
```

At the beginning

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000
2	Khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

SQL> SELECT * FROM Customers;

After ROLLBACK

ID	NAME	AGE	ADDRESS	SALARY
2	Khilan	25	Delhi	1500
3	kaushik	23	Kota	2000
4	Chaitali	25	Mumbai	6500
5	Hardik	27	Bhopal	8500
6	Komal	22	MP	4500
7	Muffy	24	Indore	10000

SQL> SELECT * FROM Customers;

TCL: RELEASE SAVEPOINT Command

TCL: RELEASE SAVEPOINT Command

- The RELEASE SAVEPOINT command is used to remove a SAVEPOINT that you have created
- The syntax for a RELEASE SAVEPOINT command is as follows.
    - RELEASE SAVEPOINT SAVEPOINT_NAME;
- Once a SAVEPOINT has been released, you can no longer use the ROLLBACK command to undo transactions performed since the last SAVEPOINT

TCL: SET TRANSACTION Command

- The SET TRANSACTION command can be used to initiate a database transaction
- This command is used to specify characteristics for the transaction that follows
  - For example, you can specify a transaction to be read only or read write
- The syntax for a SET TRANSACTION command is as follows:
  - SET TRANSACTION [READ WRITE | READ ONLY];

- **Serializability**
- Conflict Serializability

SERIALIZABILITY

Serializability

- **Basic Assumption** – Each transaction preserves database consistency
- Thus, serial execution of a set of transactions preserves database consistency
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial
  schedule.
  Different forms of schedule equivalence give rise to the notions of:

  1. **conflict serializability**
  2. **view serializability**

Simplified view of transactions

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
  - Other operations happen in memory (are temporary in nature) and (mostly) do not affect the state of the database
  - This is a simplifying assumption for analysis
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes
- Our simplified schedules consist of only **read** and **write** instructions

Conflicting Instructions

Conflicting Instructions

- Let l_i and l_j be two Instructions of transactions T_i and T_j respectively. Instructions l_i and l_j **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q

  1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict
  2. l_i = **read**(Q), l_j = **write**(Q). They conflict
  3. l_i = **write**(Q), l_j = **read**(Q). They conflict
  4. l_i = **write**(Q), l_j = **write**(Q). They conflict

- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them
  - If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule

- Serializability
- **Conflict Serializability**

CONFLICT SERIALIZABILITY

CONFLICT SERIALIZABILITY

Conflict Serializability

Conflict Serializability

- If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of non-conflicting instructions, we say that *S* and *S'* are **conflict equivalent**

- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6 – a serial schedule where T_2 follows T_1, by a series of swaps of non-conflicting instructions.
  - Swap T1.read(B) and T2.write(A)
  - Swap T1.read(B) and T2.read(A)
  - Swap T1.write(B) and T2.write(A)
  - Swap T1.write(B) and T2.read(A)

These swaps do not conflict as they work with different items (A or B) in different transactions.

- Therefore, Schedule 3 is conflict

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

Schedule 3

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Schedule 5

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Schedule 6 (serial schedule)

Conflict Serializability (Cont.)

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule $<T_3, T_4>$, or the serial schedule $< T_4, T_3 >$

Example: Bad Schedule

Example: Bad Schedule

- Consider two transactions:

Transaction 1

UPDATE accounts
SET balance = balance - 100
WHERE acct_id = 31414

Transaction 2

UPDATE accounts
SET balance = balance * 1.005

- In terms of read / write we can write these as:

Transaction 1: $r_1(A)$, $w_1(A)$ // A is the balance for acct_id = 31414

Transaction 2: $r_2(A)$, $w_2(A)$, $r_2(B)$, $w_2(B)$ // B is balance of other accounts

	A	B
(initial:)	200.00	100.00
$r_1(A)$:		
$r_2(A)$:		
$w_1(A)$:	100.00	
$w_2(A)$:	201.00	
$r_2(B)$:		
$w_2(B)$:		100.50

Schedule S

Example: Bad Schedule

- Consider schedule S:
  - Schedule *S*: $r_1(A)$, $r_2(A)$, $w_1(A)$, $w_2(A)$, $r_2(B)$, $w_2(B)$
  - *Suppose: A* starts with \$200, and account *B* starts with \$100
- Schedule *S* is very bad! (At least, it's bad if you're the bank!) We withdrew \$100 from account *A*, but somehow the database has recorded that our account now holds \$201!

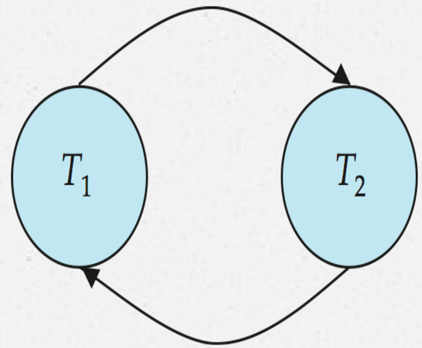
Example: Bad Schedule

- As an example, consider Schedule *T*, which has swapped the third and fourth operations from *S*:
  - Schedule *S*: $r_1(A)$, $r_2(A)$, $w_1(A)$, $w_2(A)$, $r_2(B)$, $w_2(B)$
  - Schedule *T*: $r_1(A)$, $r_2(A)$, $w_2(A)$, $w_1(A)$, $r_2(B)$, $w_2(B)$
- By first example, the outcome is the same as Serial schedule 1. But that's just a peculiarity of the data, as revealed by the second example, where the final value of A can't be the consequence of either of the possible serial schedules.
- So neither S nor T are serializable

Precedence Graph

Precedence Graph

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph**
  - A direct graph where the vertices are the transactions (names)
- We draw an arc from T_i to T_j if the two transactions conflict, and T_i accessed the data item on which the conflict arose earlier
- We may label the arc by the item that was accessed
- **Example**



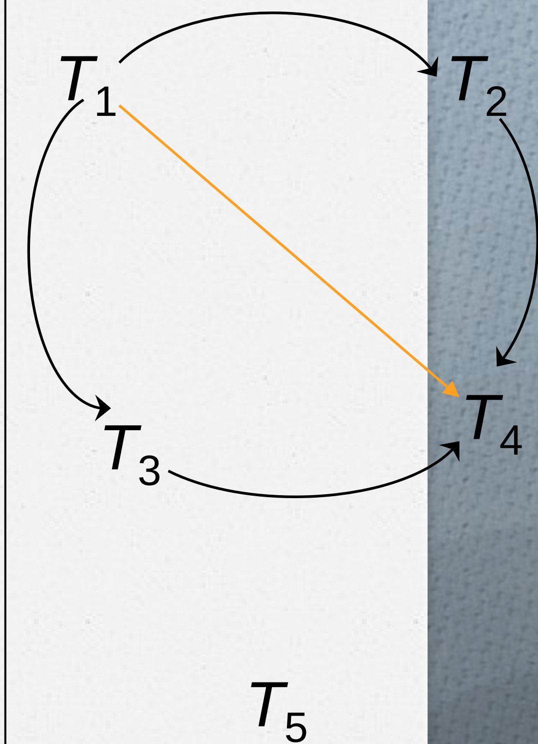
Testing for Conflict Serializability

Testing for Conflict Serializability

- Build a directed graph, with a vertex for each transaction.
- Go through each operation of the schedule.
  - If the operation is of the form $w_i(X)$, find each subsequent operation in the schedule also operating on the same data element X by a different transaction: that is, anything of the form $r_j(X)$ or $w_j(X)$. For each such subsequent operation, add a directed edge in the graph from T_i to T_j.
  - If the operation is of the form $r_i(X)$, find each subsequent *write* to the same data element X by a different transaction: that is, anything of the form $w_j(X)$. For each such subsequent write, add a directed edge in the graph from T_i to T_j.
- The schedule is conflict-serializable if and only if the resulting directed graph is acyclic.
- Moreover, we can perform a topological sort on the graph to discover the serial schedule to which the schedule is conflict-equivalent.

EXAMPLE SCHEDULE A + PRECEDENCE GRAPH

T1	T2	T3	T4	T5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

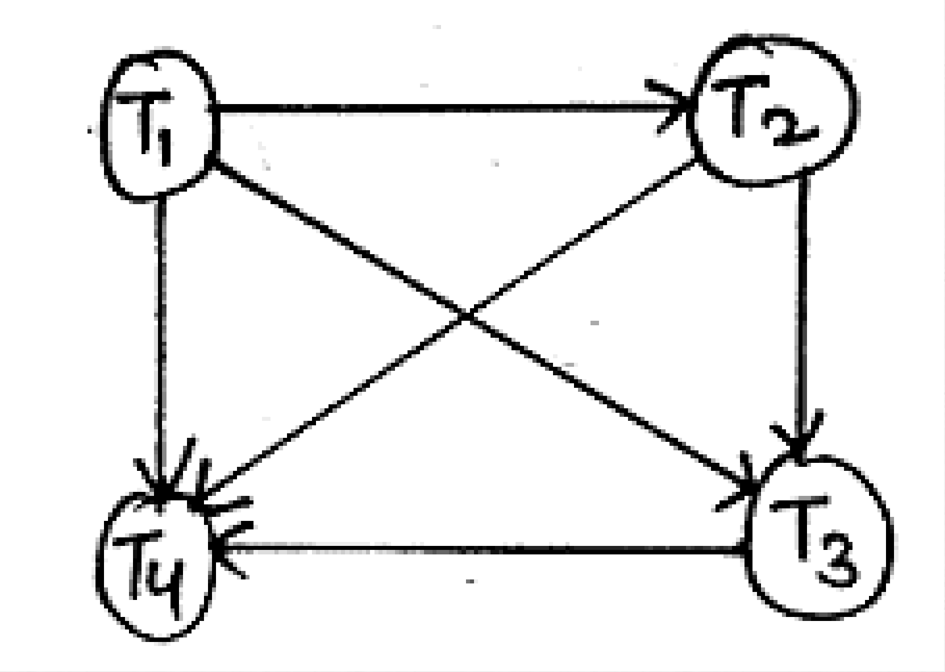


TEST FOR CONFLICT SERIALIZABILITY

- A schedule is conflict serializable if and only if its precedence graph is <u>acyclic</u>.

- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$

EXAMPLE



$$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$$

Module Summary

Module Summary

- Understood the issues that arise when two or more transactions work concurrently
- Learnt the forms of serializability in terms of conflict and view serializability
- Acyclic precedence graph can ensure conflict serializability

- Recoverability and Isolation
- Transaction Definition in SQL
- **View Serializability**

VIEW SERIALIZABILITY

VIEW SERIALIZABILITY

- Sometimes it is possible to serialize schedules that are not conflict serializable.
- View serializability provides a weaker and still consistency preserving notion of serialization.
- Let *S* and *S*´ be two schedules with the same set of transactions. *S* and *S*´ are **view equivalent** if the following three conditions are met, for each data item *Q*,

VIEW SERIALIZABILITY

1. If in schedule S, transaction T_i reads the initial value of Q, then in schedule S' also transaction T_i must read the initial value of Q.

2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j.

3. The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S'.

Serializability (Cont.)

- A schedule S is **view serializable** if it is view equivalent to a serial schedule
- Every conflict serializable schedule is also view serializable
- Below is a schedule which is view-serializable but *not* conflict serializable

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

- What serial schedule is above equivalent to?
  - T_{27}–T_{28}–T_{29}
  - The one read(Q) instruction reads the initial value of Q in both schedules and
  - T_{29} performs the final write of Q in both schedules
- T_{28} and T_{29} perform write(Q) operations called **blind writes**, without having performed a read(Q) operation
- Every view serializable schedule that is not conflict serializable has **blind writes**

View Serializability: Example 1

- Check whether the schedule is view serializable or not?
  - S : R2(B); R2(A); R1(A); R3(A); W1(B); W2(B); W3(B);
- Solution:
  - With 3 transactions, total number of schedules possible= 3! = 6
    - <T1 T2 T3>
    - <T1 T3 T2>
    - <T2 T3 T1>
    - <T2 T1 T3>
    - <T3 T1 T2>
    - <T3 T2 T1>
  - Final update on data items :
    - A : -
    - B : T1 T2 T3
    - Since the final update on B is made by T3, so the transaction T3 must execute after transactions T1 and T2.
    - Therefore, (T1,T2) → T3. Now, Removing those schedules in which T3 is not executing at last:
      - <T1 T2 T3>
      - <T2 T1 T3>

View Serializability: Example 1

View Serializability: Example 1

- Check whether the schedule is view serializable or not?
  - S : R2(B); R2(A); R1(A); R3(A); W1(B); W2(B); W3(B);
- Solution:
  - Initial Read + Which transaction updates after read?
    - A : T2 T1 T3 (initial read)
    - B : T2 (initial read); T1 (update after read)
    - The transaction T2 reads B initially which is updated by T1. So T2 must execute before T1.
    - Hence, T2 → T1. Removing those schedules in which T2 is executing before T1:
    - <T2 T1 T3>
  - Write Read Sequence (WR)
    - No need to check here
  - Hence, view equivalent serial schedule is:
    - **T2 → T1 → T3**