5 : Consistency and Transaction Processing Concepts

IT3306 – Data Management

Level II - Semester 3

Overview

- This lesson on consistency and transaction processing defines what a transaction is and its properties.
- Here we look at schedules and serializability.
- Finally, we explore transaction support in SQL and maintaining consistency in NoSQL.

Intended Learning Outcomes

At the end of this lesson, you will be able to;

- Understand what transaction processing is
- Define properties of transactions
- Understand schedules and serializability
- Identify different types of serializability techniques
- Explain transaction support in SQL
- Understand consistency in NoSQL

List of subtopics

5.1. Introduction to Transaction Processing

- 5.1.1. Single-user systems, multi-user systems and Transactions
- 5.1.2. Transaction states
- 5.1.3. Problems in concurrent transaction processing, introduction to concurrency control, DBMS failures, introduction to data recovery

5.2. Properties of Transactions

- 5.2.1. ACID properties, levels of isolation

5.3. Schedules

- 5.3.1. Schedules of Transactions
- 5.3.2. Schedules Based on Recoverability

5.4. Serializability

- 5.4.1. Serial, Nonserial, and Conflict-Serializable Schedules

List of subtopics

5.4.2. Testing for Serializability of a Schedule
5.4.3. Using Serializability for Concurrency Control
5.4.4. View Equivalence and View Serializability
5.5. Transaction Support in SQL
5.6. Consistency in NoSQL
5.6.1. Update Consistency
5.6.2. Read Consistency
5.6.3. Relaxing Consistency
5.6.4. CAP theorem
5.6.5. Relaxing Durability and Quorums
5.6.6. Version Stamps

5.1.1. Single-user systems, Multi-user systems and Transactions

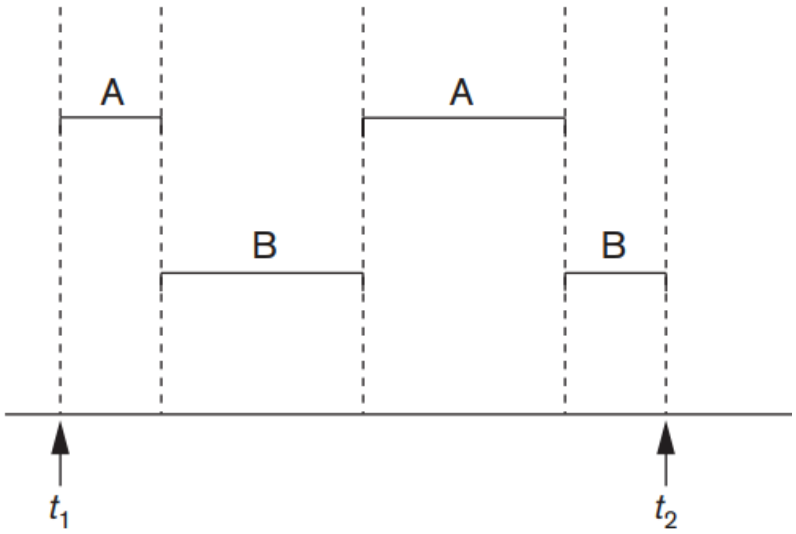
Databases can be classified based on the number of concurrent users.

- **Single - User Systems -** Database can be accessed by one user at a time. Most commonly these are used by personal computer systems.
- **Multi - User Systems -** Database can be accessed by many users at the same time. This is the concurrent use of database. Database systems used in airline reservation systems, supermarkets, hospitals, banks and stock exchange systems are accessed by hundreds or thousands of users at the same time.

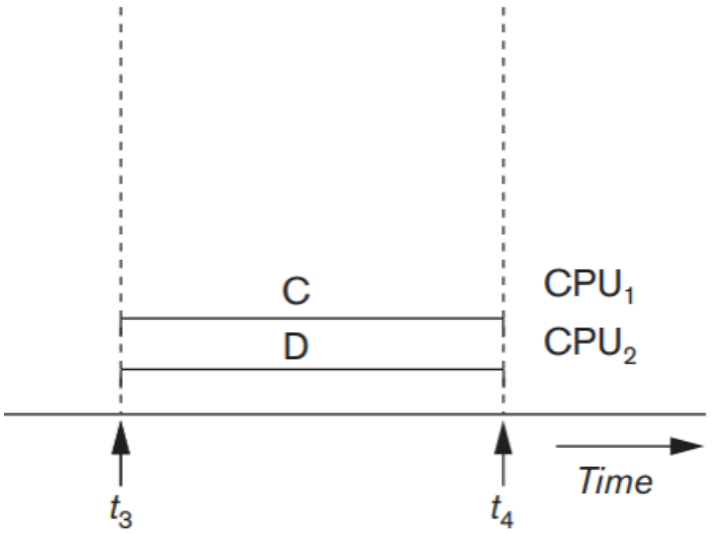
5.1.1. Single-user systems, Multi-user systems and Transactions

- **Multiprogramming** is the concept behind this simultaneous access of the database by several users. In multiple programming, operating system of the computer is allowed to execute multiple programmes at a time.
- In Central Processing Unit (CPU), only one process can be executed at a time.
- Hence, in multiprogramming systems, CPU executes set of commands from one process and then suspends it and again executes a set of commands from another process.
- A suspended process will resume again from the point where it was suspended when it gets the chance to use the CPU again. This pattern continues to keep running multiple processes.
- This way, the actual process of concurrent execution is **interleaved**.

5.1.1. Single-user systems, Multi-user systems and Transactions



(i) Interleaved processing



(ii) Parallel processing

5.1.1. Single-user systems, Multi-user systems and Transactions

- When interleaving is not allowed, several problems may occur as given below:
  - Some processes have to remain idle if the active process wants to execute I/O operations, like reading a block from disk. The reason is that it is not allowed to switch CPU to execute another process.
  - Some delays could occur since some processes have to wait until long processes finish execution.
- When there are multiple CPUs present in a computer, **parallel processing** can take place as shown in C,D of the figure (ii) in the previous diagram.
- In general, the theories on concurrency control of DBMS are based on **interleaved concurrency** .

5.1.1. Single-user systems, Multi-user systems and Transactions

- A ***transaction*** is an executing program that forms a logical unit of database processing. Therefore, **a transaction is defined as a logical unit of database operations.**
- A transaction may consist of one or more database access operations. These include insertion, deletion, modification (update), or retrieval operations.
- The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL.

5.1.1. Single-user systems, Multi-user systems and Transactions

- The transaction boundaries can be specified with explicit ***begin transaction*** and ***end transaction*** statements in an application program.
- In this case, all database access operations between these two statements are considered as forming one transaction.
- If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **r*ead-only transaction****;* otherwise, it is known as a ***read-write transaction****.*

5.1.1. Single-user systems, Multi-user systems and Transactions

- Concurrency control and recovery mechanisms are mainly concerned with the database commands in a transaction.
- Transactions submitted by the various users may execute concurrently and may access and update the same database items.
- If the concurrent execution is uncontrolled, it may lead to problems, such as an inconsistent database.

5.1.2. Transaction States

- The system needs to keep track of when each transaction starts, terminates, and commits/aborts for recovery purposes.
- Therefore, following operations need to be tracked by the recovery manager of the DBMS.
  - **BEGIN_TRANSACTION**

    Marks the start of executing a transaction.
  - **READ or WRITE**

    Defines read or write operations on the database
  - **END_TRANSACTION**

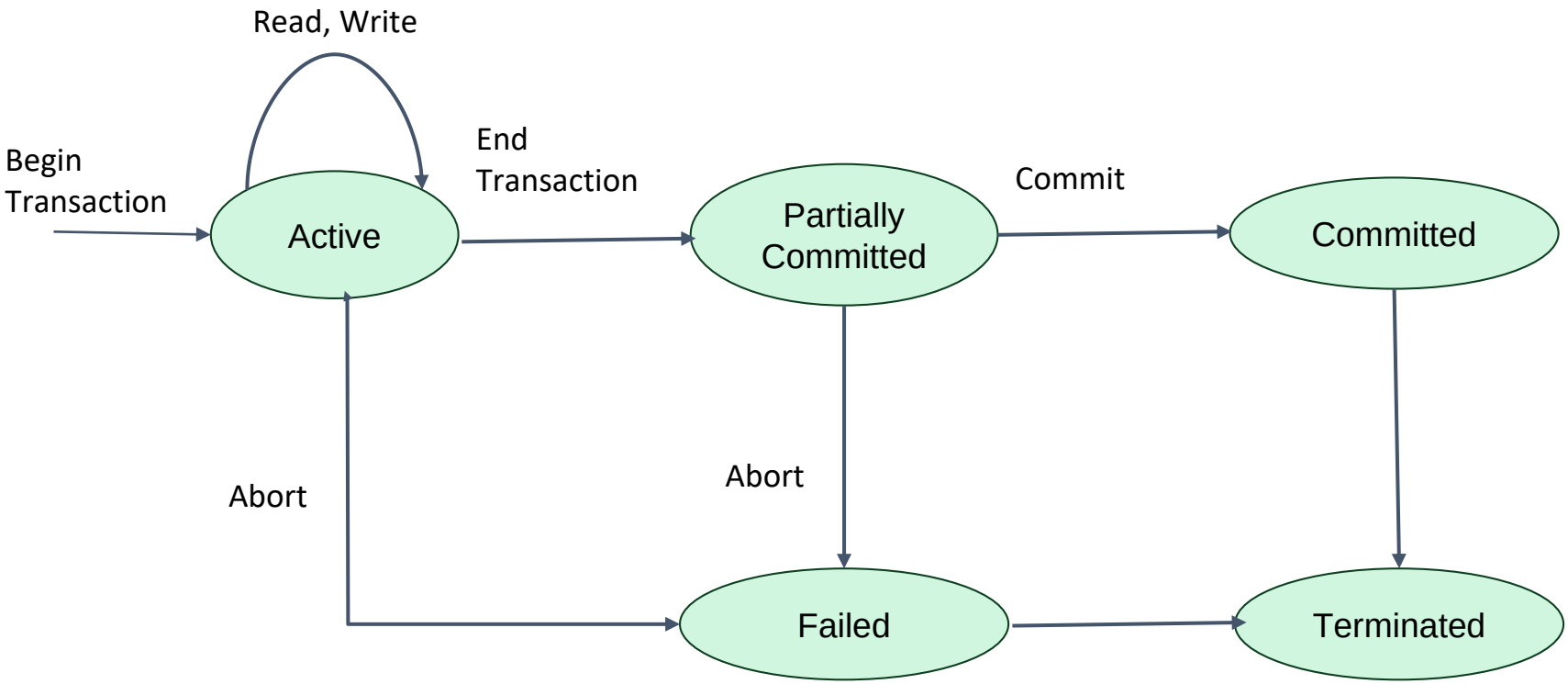
    Indicates the end of all READ/WRITE operations and characterizes the end of transaction execution.

5.1.2. Transaction States

- At the end of a transaction, it might be necessary to check whether the transaction is committed or aborted.
  - **COMMIT_TRANSACTION**
    - Indicates successful completion of a transaction and capability to safely commit the updates resulted by the transaction to the database. These updates made to the database will not be undone.
  - **ROLLBACK or ABORT**
    - Indicates the end of an unsuccessful transaction. Any change/ update to the database made by the aborted transaction must be undone.

5.1.2. Transaction States

Following state transition diagram illustrates how a transaction moves through its execution states.



5.1.2. Transaction States

- Just after the start of a transaction, it goes into active state. At this state, the transaction executes its read and write operations**.**
- Once the transaction ends, it shifts to partially committed state. Some concurrency control protocols, and additional checks might be applied to find out whether the transaction can be committed or not.
- Further, some recovery protocols are essential to assure that no system failures will occur, and changes resulted from the transactions can be permanently recorded to the database.

5.1.2. Transaction States

- If the above checks are successful, it goes to the committed state where all the updates are successfully recorded into the database. Otherwise, the transaction will be aborted and goes to the failed state, where all the updates should be rolled back.
- Also, a transaction might go to the failed state if it was aborted during the active state.
- In the terminated state, the transaction leaves the system.
- Failed or aborted transactions might start again automatically or through a resubmission done by the user.

5.1.3. Problems in Concurrent Execution

Example Transaction

Account balance of A (X) is 1000;

Account balance of B (Y) is 2000;

Transaction T1 - Rs.50 is withdrawn from A and deposited in B.

Transaction T2 - Rs.100 deposited to account A.

T1

```
read_item(X)
X:= X-N;
write_item (X);
read_item(Y);
Y:= Y+N;
write_item(Y);
```

T2

```
read_item(X)
X:= X+M;
write_item (X);
```

T1	T2
A = 1000	A = 1000
A – 50= 950	A + M = 1100
A = 950	A = 1100
B = 2000	
B = 2000 + 50	
B = 2050	

After T1 and T2 completed without interleaving, the final values of A and B should be, A=**1050** and B=**2050.**

5.1.3. Problems in Concurrent Execution

But, when these two transactions execute in interleaved fashion, following problems can be occurred.

1. Lost Update Problem
2. Temporary Update (Dirty Read) Problem
3. Incorrect Summary Problem
4. Unrepeatable Read Problem

Let’s discuss these problems in detail.

5.1.3. Problems in Concurrent Transaction Processing

The Lost Update Problem.

- When two interleaved transactions access the same item from the database, it would result in an incorrect value for that item.
- Assume the two transactions T_1, T_2 (example in slide 18) have been submitted in an interleaved fashion, as shown in the table.

T1	T2
read_item(X); *1000*	
X = X - N	
	read_item(X); *1000*
	X = X +M;
write_item(X); *950*	
read_item(Y); *2000*	
	write_item(X); *1100*
Y = Y + N;	
write_item(Y); *2050*	

Item X has an incorrect value because its update by T1 is lost (overwritten).
Thus, the final value of X will be 1100 with respect to this execution instead of 1050.

5.1.3. Problems in Concurrent Transaction Processing

The Lost Update Problem - Example

Suppose there are 2 trains X and Y, which has 80 reservations for X and 100 reservations for Y. (Refer the next slide for tabular representation)

- One person submits a cancellation of 8 seats (N=8) for X train, and do a reservation of 8 seats on train Y. At the same time another person submits a reservation of 2 seats (M = 2) for train X.
- At the end of these two processes, resulting reservations should be X=(80-8+2)= 74 and Y=(100+8)=108.

5.1.3. Problems in Concurrent Transaction Processing

The Lost Update Problem -Example

T1	T2
READ(X)	
X=X-N	
	READ(X)
WRITE(X)	
	X=X+M
	WRITE(X)
READ(Y)	
Y=Y+N	
WRITE(Y)	

T1	T2
X=80	
X=80-8	
	X=80
X=72	
	X=80+2
	X=82
Y=100	
Y=100+8	
Y=108	

Still the value of X is 80, because the change done by T1 is not yet written.

- But, after the execution, the results (X=82 and Y=108) does not match with expected calculations (X=74 and Y=108).
- The resulting X value is incorrect **because the update done by T1 for X is lost** and T2 gets the X value directly from the DB.

5.1.3. Problems in Concurrent Transaction Processing

The Temporary Update (or Dirty Read) Problem.

- Dirty reads happen when one transaction updates a database item and, however that particular transaction has failed to complete due to some reason.
- In the meantime, some other transaction has accessed the same database item which is temporarily updated and has not yet rolled back to its original value. This is a dirty read.

5.1.3. Problems in Concurrent Transaction Processing

The Temporary Update (or Dirty Read) Problem

T1	T2

read_item(X);
X = X - N;
write_item(X);	

	read_item(X);
	X = X + M;
	write_item(X);

read_item(Y);	
rollback;	

- In the given table, transaction T1 has updated the value of X, and then transaction T2 has read the updated value of X.

- However, at some point, T1 fails and by that time T2 has already accessed the temporary updated value of X, which would eventually roll back (changed) to its old value.
- The value accessed by T2 is a dirty read, because it has read a value modified by an incomplete transaction which is not committed.

5.1.3. Problems in Concurrent Transaction Processing

The Temporary Update (or Dirty Read) Problem

Example

X= 80; Y=100;, N=5;M=4;

T1	T2
READ(X);	
X=X-N;	
WRITE(X);	
	READ(X);
	X=X+M;
	WRITE(X);
READ(Y);	
ROLLBACK;	

T1	T2
80	
X=80-5	
75	
	75
	X=75+4
	79
100	
ROLLBACK	

X value should be 80 when T1 is rolled back. But, T2 has read X from the temporary update done by T1.

When T1 is rolled back, X value is being 80 again. But T2 has read an incorrect value from an uncommitted transaction.

5.1.3. Problems in Concurrent Transaction Processing

The Incorrect Summary Problem

- Take an instance where one transaction is getting the aggregated summary of database items and another transaction is updating the values of the same database items. Both these transactions are running in an interleaved manner.
- This results in some values being not updated yet and some values being already updated when they are getting read by the aggregate function.
- Hence, gives a wrong summary.

5.1.3. Problems in Concurrent Transaction Processing

The Incorrect Summary Problem

T1	T3

	sum = 0;
	read_item(A);
	sum = sum + A;

read_item(X);
X = X - N;
write_item(X);	

	read_item(X);
	sum = sum + X;
	read_item(Y);
	sum = sum + Y;

read_iten(Y);
Y = Y + N;
write_item(Y);	

T1 changes the value of X by subtracting N.
T3 reads X (after N is subtracted)
T3 reads Y (before N is added).
T1 changes the value of Y by adding N.
The change of Y value done by T1 is not considered when calculating the sum.
Therefore, the resulted sum is wrong.

5.1.3. Problems in Concurrent Transaction Processing

The Incorrect Summary Problem

X= 80; Y=100;, N=5; M=4; A=5;

T1	T3
	SUM=0;
	READ(A);
	SUM+=A;
READ(X);	
X=X-N;	
WRITE(X);	
	READ(X);
	SUM+=X;
	READ(Y);
	SUM+=Y;
READ(Y);	
Y=Y+N;	
WRITE(Y);	

T1	T3
	0
	5
	5
80	
80-5	
75	
	75
	5+75
	100
	80+100
100	
100+5	
105	

T3 reads X after it is updated by T1. The correct value of X is taken for the sum. But T3 reads Y before it is getting updated and hence read an incorrect value for the sum.

The correct sum after reading Y should be 80 + 105.
But instead it gives 80 + 100 since y is read as 100 instead of 105.

5.1.3. Problems in Concurrent Transaction Processing

The Unrepeatable Read Problem.

- This occurs when one transaction reads a particular database item twice and get two different values.
- The reason is some other transaction has made an update on the very same database item between the two reads.

5.1.3. Problems in Concurrent Transaction Processing

The Unrepeatable Read Problem -Example

X= 80

T1	T2
READ(X)	
	READ(X) X=X-5 WRITE(X)
READ(X)	

T1	T2
80	
	80 80-5 75
75	

T1 gives 2 different values, when reading the same data item.

5.1.3. DBMS Failures

- A Transaction is considered as **committed**, if all the operations of the submitted transaction are successfully executed and the effect of that particular transaction on the database items is permanently recorded.
- If a transaction fails to complete successfully, it is considered as **aborted,** where the database gets no effect.
- When some operations fail to execute in the transaction, the previous, successful operations relevant to that transaction should be undone to make sure there is no effect to the database.

5.1.3. DBMS Failures

- **A computer failure (System Crash).**

  Occurs due to hardware, software, or network error in the computer system during transaction execution.

- **A transaction or system error.**

  Occurs due to the errors in operation such as integer overflow or division by zero. Some other reasons are inaccurate parameters, logical programming errors and interruptions from the user.

5.1.3. DBMS Failures

- **Local errors or exception conditions detected by the transaction.**
  - Some exceptions in the programme may cause cancellation of a transaction.
  - For instance, the data might not be available to complete the transaction or the existing values do not meet the required conditions.
  - As an example, we cannot withdraw money from an account which does not have a sufficient balance.

5.1.3. DBMS Failures

- **Concurrency control enforcement.**
  - Failures may take place due to the enforcement of concurrency control.
- **Disk failure.**

  - Disk failures may occur while performing read/write operations.

  - Due to read or write malfunction, data in some disk blocks may lose.

  - Sometimes this can happen as a result of a crash in disk read/write head.

5.1.3. DBMS Failures

- **Physical problems and catastrophes.**

  - This includes numerous problems such as power loss, failure in air-conditioning, natural disasters, theft, sabotage and mistakenly overwriting disks or tapes etc.

5.2. Properties of Transactions

ACID properties

ACID are the properties of transactions which are imposed by concurrency control and recovery methods of the DBMS.

ACID stands for

i) A – Atomicity

ii) C – Consistency

iii) I – Isolation

iv) D – Durability

A detailed description of each property is explained in the upcoming slides.

5.2. Properties of Transactions

i) Atomicity

- A transaction is an atomic unit with regards to transaction processing. This infers that a transaction either ought to be executed completely or not performed at all. Thus, the atomicity property necessitates that a transaction executes to its completion.
- The transaction recovery subsystem of a DBMS guarantees atomicity. In the event that a transaction fails to finish (for example, a system crash occurs amidst its execution), the recovery strategy should fix any impacts of the transaction on the database through undo/redo operations. Write operations of a committed transaction must be written to disk.

5.2. Properties of Transactions

Consider the following transaction T1 as an example to illustrate the properties of transactions.

Transfer 50 from account A to account B.

A = 1000; B =2000;

T1:

Begin read (A);

A=A-50;

write (A);

read (B);

B=B+50

write (B)

End;

5.2. Properties of Transactions

Example for Atomicity

- Definition :- Either a transaction is performed in its entirety or not performed at all.

- When considering T1(in previous slide):
  If a transaction failure occurs after write (A), but before write (B);
  then A=950; B=2000; 50 is lost.
  Data is now inconsistent with A+B = 2950 instead of 3000.

- Therefore, the transaction should either be fully executed, or else data should reflect as if the transaction never started at all.

5.2. Properties of Transactions

ii) Consistency

- A transaction should be completely executed from beginning to end without getting interfered by other transactions to preserve the consistency. A transaction leads the database from one consistent state to another.
- A database state is a collection of all the data values in the database at a given point.
- The conservation of consistency is viewed as the responsibility of the developers who compose the programs and of the DBMS module that upholds integrity constraints.

5.2. Properties of Transactions

ii) Consistency

- A consistent state of the database fulfils the requirements indicated in the schema and other constraints on the database that should hold.
- If a database is in a consistent state before executing the transaction, then it will be in a consistent state after the complete execution of the transaction (assuming that no interference occurs with other transactions).

5.2. Properties of Transactions

Example for Consistency

- Definition :- Take database from one consistent state to another.

- When considering T1:
  initially, A = 1000 and B = 2000.
  A + B = 3000.

  After the transaction T1, A = 950 and B = 2050
  A + B = 3000.

  Thus, the value of A+B = 3000 should be the same before and after the transaction.

5.2. Properties of Transactions

iii) Isolation

- During the execution of a transaction, it should appear as if it is isolated from other transactions even though there are many transactions happening concurrently.
- The execution of a transaction should not interfere with other transactions executing simultaneously.
- The isolation property is authorized by the concurrency control subsystem of the DBMS.
- In the event that each transaction doesn't make its write updates apparent to other transactions until it is submitted, one type of isolation is authorized that takes care of the temporary update issue.

5.2. Properties of Transactions

Example for Isolation

- Definition :- Updates not visible to other transactions until committed

- When considering T1:
  - Initially, A = 1000 and B = 2000.

  - Between WRITE(A) and WRITE (B) of T1, if another transaction performs READ(A) and READ(B) operations, the values seen is inconsistent (A+B=2950).

5.2. Properties of Transactions

Levels of Isolation

Before talking about isolation levels, let's discuss about database locks.

Database Locks

A database lock is used to "lock" data in a database table so that only one transaction/user/session may edit it. Database locks are used to prevent two or more transactions from changing the same piece of data at the same time.

5.2. Properties of Transactions

Levels of Isolation

There have been attempts to define the level of isolation of a transaction.

- **Level 0 (zero) isolation** (known as Read Uncommitted) - If a transaction does not overwrite the dirty reads of higher-level transactions.
- **Level 1 (one) isolation** (known as Read Committed) **-** If a transaction has no lost updates.
- **Level 2 isolation** (known as Repeatable Read) - If a transaction has no lost updates and no dirty reads.
- **Level 3 Isolation** / **True isolation** (known as Serializable Read) - If a transaction has no lost updates, no dirty reads and no repeatable reads.

5.2. Properties of Transactions

Levels of Isolation

Example for Level 0 (zero) isolation

T1	T2
update employee set salary = salary - 100 where emp_number = 25	
	select sum(salary) from employee
	Commit;
Rollback;	

T1 updates the salary of one employee by subtracting Rs. 100.

T2 requests the sum of salaries of all employees. Then T2 ends.

T1 rolls back, invalidating the results from T2, since T2 reads a value updated by an uncommitted transaction.

5.2. Properties of Transactions

Levels of Isolation

Example for Level 0 (zero) isolation

Therefore, in level 0 Isolation it allows a transaction to read uncommitted changes. This is also known as a dirty read, since the new transaction may display results that are later rolled back with respect to the older transaction.

5.2. Properties of Transactions

Levels of Isolation

Example for Level 1 isolation

T1	T2
update employee set salary = salary - 100 where emp_number = 25;	
	select sum(salary) from employee where emp_number < 50;
rollback	
	commit

T1 updates the salary of employee with emp_number 25 by subtracting Rs. 100.

T1 does not commit its update. T2 queries to get the sum of salaries of the set of employees with emp_number less than 50 in employee table. However, T1's update is not captured in this query as it is not committed. This is because Level 1 isolation does not consider uncommitted transactions. T1 rolls back. With Level 1 isolation, dirty reads are prevented.

5.2. Properties of Transactions

Levels of Isolation

Example for Level 2 isolation

Explanation of the example is given in the next slide.

T1	T2
select sum(salary) from employee where emp_number < 25	
	update employee set salary= salary- 100 where emp_number = 22
	commit transaction
select sum(salary) from employee where emp_number < 25	
commit transaction	

5.2. Properties of Transactions

Levels of Isolation

Example for Level 2 isolation

In the example in previous slide;
T1 queries to get the sum of salaries of employees whose emp_number is less than 25. T2 updates the salary of the employee whose emp_number is 22. Then T1 executes the same query again.
If transaction T2 modifies and commits the changes to the employee table after the first query in T1, but before the second one, the same two queries in T1 would produce different results. Isolation level 2 blocks transaction T2 from executing. It would also block a transaction that attempted to delete the selected row. Thus, lost updates and dirty reads are avoided.

5.2. Properties of Transactions

Phantoms

- If a database table includes a record which was not present at the start of a transaction but is present at the end then it is called a phantom record.
- For example, If transaction T2 enters a record to a table that transaction T1 currently reads (the record also satisfies the filtering conditions used in T1), then that record is a phantom because it was not there when T1 started but is there when T1 ends.
- If the equivalent serial order is T1 followed by T2, then the record should not be seen. But if it is T2 followed by T1, then the phantom record should be in the result given to T1.

5.2. Properties of Transactions

Levels of Isolation

Example for Level 2 isolation

Consider the following example on phantom reads

T1	T2
select * from employee where salary>45000	
	insert into employee (emp_number, salary) values (19, 50000)
	commit transaction
select * from employee where salary>45000	
commit transaction	

5.2. Properties of Transactions

Levels of Isolation

Example for Level 2 isolation (Phantom reads in Level 2 Isolation)

In the example given in the previous slide;
T1 retrieves the rows from employee table where salaries are more than 45000. Then T2 inserts a row that meets the criteria given in T1 (an employee whose salary is greater than 45000) and commits. T1 issues the same query again. The number of rows retrieved for the same select query in T1 are different when the isolation level is 2.

Total no. of records retrieved by executing second select statement = total no. of records retrieved by first select statement is +1.
This creates a phantom. Phantoms occur when one transaction reads a set of rows that satisfy a search condition, and then a second transaction modifies those data. If the first transaction repeats the read with the same search conditions, it obtains a different set of rows.
In the above example, T1 sees a phantom row in the second select query.

5.2. Properties of Transactions

Levels of Isolation

Example for Level 2 isolation (Phantom reads in Level 2 Isolation)

The issue on phantom reads in Level 2 isolation, is prevented in Level 3 isolation which is also known as Serializable Read. Level 3 isolation has no lost updates, no dirty reads and no repeatable reads in transactions.

Let’s look at how this mitigation is done in Level 3 isolation. Let’s utilise the same example we used in phantom reads to discuss the mitigation mechanism.

5.2. Properties of Transactions

Levels of Isolation

Example for Level 3 isolation

Explanation of the example is given in the next slide.

T1	T2
select * from employee where salary>45000	
	insert into employee (emp_number, total) values (19, 50000)
	commit transaction
select * from employee where salary>45000	
commit transaction	

5.2. Properties of Transactions

Levels of Isolation

Example for Level 3 isolation

In the table shown in previous slide;

T1 retrieves a set of rows where salaries are more than 45000, and holds a database lock. Then T2 inserts a row that meets this criteria for the query in T1, but must wait until T1 releases its lock (locked items are only accessed by the transaction which holds the lock). Thereafter, T1 makes the same query and gets same results (unlike what we discussed in slide 54). Then, T1 ends and releases its lock. Now T2 gets its lock, inserts new row, and ends.
This prevents phantoms.
In Level 3 isolation, database locks are utilized to avoid phantom reads (slide 45).

5.2. Properties of Transactions

Snapshot isolation

- Another kind of isolation is called snapshot isolation, which is utilized in some commercial DBMSs. Several concurrency control techniques depend on this.

- At the start of a transaction, it sees the data items that it reads based on the committed values of the items in the database snapshot (or database state).

- Due to this property, it ensures that the phantom read problem does not occur (since the database transaction will only see the records that were committed at the time the transaction starts).

- Therefore, any insertions, deletions, or updates that occur after starting the transaction, will not be seen by it.

5.2. Properties of Transactions

Snapshot isolation

Example for Snapshot isolation

T1	T2
SELECT * FROM employee ORDER BY empID;	
	INSERT INTO employee (empID, empname) VALUES(600, 'Anura'); COMMIT;
SELECT * FROM employee ORDER BY empID;	
	INSERT INTO employee (empID, empname) VALUES(700, 'Arjuna')
COMMIT;	
SELECT * FROM employee ORDER BY empID;	

empID	empname
100	Upul
200	Manjitha

This is the output of the first select query of T1. It only generates the data that is available in the current snapshot.

5.2. Properties of Transactions

Snapshot isolation

T1	T2
SELECT * FROM employee ORDER BY empID;	
	INSERT INTO employee (empID, empname) VALUES(600, 'Anura'); COMMIT;
SELECT * FROM employee ORDER BY empID;	
	INSERT INTO employee (empID, empname) VALUES(700, 'Arjuna')
COMMIT;	
SELECT * FROM employee ORDER BY empID;	

empID	empname
100	Upul
200	Manjitha

The second select statement of T1 produces the same result as the first select statement, because T1 has not committed yet. The snapshot taken by T1 remains without changing until it commits.

5.2. Properties of Transactions

Snapshot isolation

T1	T2
SELECT * FROM employee ORDER BY empID;	
	INSERT INTO employee (empID, empname) VALUES(600, 'Anura'); COMMIT;
SELECT * FROM employee ORDER BY empID;	
	INSERT INTO employee (empID, empname) VALUES(700, 'Arjuna')
COMMIT;	
SELECT * FROM employee ORDER BY empID;	

empID	empname
100	Upul
200	Manjitha
300	Anura

This is the output of the third select query of T1.
Since T1 is committed, a new snapshot is taken for further queries. As the first insert query of T2 is now committed, that inserted row can be seen in the snapshot with the updated value of 'Aruna'. However, 'Arjuna' does not get updated as that insertion query has not committed yet in T2.

5.2. Properties of Transactions

Snapshot isolation

T1	T2
SELECT * FROM employee ORDER BY empID;	
	INSERT INTO employee (empID, empname) VALUES(600, 'Anura'); COMMIT;
SELECT * FROM employee ORDER BY empID;	
	INSERT INTO employee (empID, empname) VALUES(700, 'Arjuna')
COMMIT;	
SELECT * FROM employee ORDER BY empID;	

empID	empname
100	Upul
200	Manjitha
300	Anura

As explained in the previous slide, third select is based on a new snapshot. The new snapshot contains all the commits up to now, which includes the insertion of Anura. Therefore it shows all three records.

However, the insertion of Arjuna is not seen because this second insertion in T2 is not committed.

5.2. Properties of Transactions

iv) Durability

- Durability or permanency means, once the changes of a transaction are committed to the database, those changes must remain in the database and should not be lost.

- Therefore, this property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs.

- These updates now become permanent and are stored in non-volatile memory. Therefore, effects of the transaction are never lost.

5.2. Properties of Transactions

Example for Durability

- Definition : Changes must never be lost because of subsequent failures (eg: power failure)
- In the transaction T1, if transaction failure occurs after write (A), but before write (B);

  To recover the database,

  i. We must remove changes of partially done transactions. Therefore, the change done on A should be rolled back. (before crash, A was 950. Then it needs to be rolled back to 1000)
  ii. We need to reconstruct completed transactions. If the system fails after the commit operation of a transaction, but before the data could be written on to the disk, then that transaction needs to be reconstructed.

The database should keep all its latest updates even if the system fails. If a transaction commits after updating data, then the database should have the modified data.

Activity

Mark the following as true or false.

1. Single - User System databases can be accessed only by one user at a time.
2. Multi - User Systems enables concurrent use of the database.
3. In Central Processing Unit (CPU), many processes can be executed at the same time.
4. Multiprogramming is the concept behind simultaneous access of the database by several user.
5. In multiprogramming systems, CPU executes one command from one process and then suspends it and then executes a set of commands from another process.

Activity

Identify the problem that would result in the following transaction processing.

T1	T2
read (x)	
x = x - n	
	read (x)
	x = x + m
write (x)	
read (y)	
	write (x)
y = y + n	
write (y)	

Activity

Identify the problem that would result in the following transaction processing.

T1	T2
read (x)	
x = x - n	
write (x)	
	read (x)
	x = x + m
	write (x)
	commit
read (y)	
abort	

Activity

Identify the problem that would result in the given transaction processing.

T1	T2
	sum = 0
	read (a)
	sum = sum + a
read (x)	
x = x - n	
write (x)	
	read (x)
	sum = sum + x
	read (y)
	sum = sum + y
read (y)	
y = y + n	
write (y)	

Activity

State the main problems in concurrent transaction processing.

a. _____

b. _____

c. _____

d. _____

Activity

Fill in the blanks using the correct option.

1. A transaction is considered as _____ , if all the operations of the submitted transactions are successfully executed and the effect of that particular transaction on the database items are permanently recorded.
2. If a transaction did not successfully complete, it is considered as _____.
3. A transaction is a _____ unit of database operations.
4. When some operations get failed to execute in the transaction, the previous successful operations should be _____ to make sure there is no effect to the _____ .

Activity

Drag and drop the matching words for the sentence

1. Problems caused by hardware, software, or network error that occurs in the computer system during transaction execution. – _____
2. Occurs due to the errors in operation such as integer overflow or division by zero. – _____
3. Occurs due to some exception in the programme cause the cancellation of a transaction – _____
4. Occurs due to read or write malfunction and data in some disk blocks may get lost – _____
5. Problems such as power loss, failure in air-conditioning, natural disasters, theft, sabotage, mistakenly overwriting disks or tape, and mounting of a wrong tape by the operator – _____

Activity

item table=>

item_no	1	2	3	4	5	6	7
price	5500	2500	4500	2000	3250	4900	2100

A list of item numbers and their prices are given in the above table. After the two transactions T1, T2 were executed on the above item table, the output was 14,500.

What can be the least possible isolation level used in T2?

T1	T2
update item set price = price - 1000 where item_no = 2;	
	select sum(price) from item where item_no < 5;
rollback	
	commit

Activity

State the four main properties of a transaction.

1. _____
2. _____
3. _____
4. _____

5.3 Schedules

Schedules of Transactions

- The arrangement or order of operations in a transaction is named as a ***schedule***.

$$S = T_1, T_2, T_3, ..., T_n$$

- A schedule can be ***interleaved***, which executes operations from different transactions.
- But, every transaction T_i that appears in the Schedule S, should follow the order of operations as if the transaction executes in isolation.

5.3 Schedules

Schedules of Transactions

- In this slide set, we will be using a set of notations for the operations included in a transaction and to identify the transaction number we will be adding a subscript.
- Following are the notations and their descriptions, that we use in this slide set.

b	begin_transaction
r	read_item
w	write_item
e	end_transaction
c	commit
a	abort

5.3 Schedules

$S_a : r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$

The above schedule can be arranged into a tabular form by separating the operations into 2 transactions T1 and T2.

T1	T2
r(X)	
	r(X)
w(X)	
r(Y)	
	w(X)
w(Y)	

5.3 Schedules

- **Schedules of Transactions**

  If two operations in a schedule have the following properties, it is known as a ***conflict***.

  1. Operations are from different transactions.
  2. Do the operation on same data item.
  3. At least one of the two operations is a write (insert, update, delete)

Ex: $r_1(X)$ and $w_2(X)$ -> conflict

$w_1(X)$ and $w_2(X)$ -> conflict

$r_1(X)$ and $r_2(X)$ -> not a conflict

$r_1(X)$ and $w_2(Y)$ -> not a conflict

5.3 Schedules

- **Schedules Based on Recoverability**
  - After a system failure, we have to recover the system.
  - There are some schedules, which are easy to recover and some of the schedules cannot be recovered.
  - We are going to characterize the schedules based on recoverability.
  - Recoverable schedules:- **Rolling back should not be needed once a transaction is committed.**

5.3 Schedules

- **Schedules Based on Recoverability**
  - A schedule with the following properties is known as a recoverable schedule.

1. **Every transaction T1 , in the schedule S should not be committed until all transactions T2 which has written values that T1 read is committed.**
2. **T2 should not have been aborted before T1 reads item X.**
3. **In between T2 writes X and T1 reads X, there should be no transactions that write X.**

5.3 Schedules

- **Schedules Based on Recoverability (Example)**

S'= $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; c_2; a_1;

S' is non recoverable.

Reason: T_2 reads item X written by T_1, but T_2 commits before T_1 commits. The problem occurs by T_1 aborting after the c_2 operation. Then the value of X that T_2 read is no longer valid and T_2 cannot be aborted as it has already been committed, which leads to a schedule that is **not recoverable**.

For the schedule to be recoverable, the c_2 operation in S' must be postponed until T_1 commits.

T1	T2
r(X)	
w(X)	
	r(X)
r(Y)	
	w(X)
	c
a	

5.3 Schedules

Schedules Based on Recoverability

- As the name implies, ***Cascading rollback*** is rolling back an uncommitted transaction due to the fact that it performs a read on a failed transaction.

$$S'' = r_1(X);\ w_1(X);\ r_2(X);\ r_1(Y);\ w_2(X);\ w_1(Y);\ a_1;\ a_2;$$

In the above example, T_2 is also aborted since T_1 aborted. The reason here is that, T_2 reads the X value from T_1.

- A schedule where cascading rollback does not happen is known as a ***cascadeless schedule***.

5.3 Schedules

Schedules Based on Recoverability

- In a ***Strict schedule***, no item can be read or written by a transaction until the commit operation of the last transaction which performed the write of that item occurs.
- In simple terms, in a strict schedule, it is not possible to read or write a value which is written by an uncommitted transaction.

5.4 Serializability

Serial, Nonserial, and Conflict-Serializable Schedules

- ***Serializable schedules*** are the schedules which are considered as correct when executing in the interleaved fashion.
- If all the transactions of a schedule can perform all its operations consecutively, it is known as ***serial.***

$$S' = r_1(X); w_1(X); r_1(Y); w_1(Y); c_1; r_2(X); w_2(X); c_2;$$

- In S' schedule, all the operations of T1 are completed and then operations of T2 are started. Therefore, S' is a serial schedule.
- Theoretically, we can say that a serial schedule is always correct, since it performs one transaction after the commit/abort of the previous transaction.

5.4 Serializability

Serial, Nonserial, and Conflict-Serializable Schedules

- But there are problems in serial schedules:
  - Limits the concurrency execution.
  - It may be time consuming, because while one transaction waits for I/O operation to be completed, it does not allow to execute another transaction.
  - If one transaction is long, the next transaction has to wait a considerable amount of time until the previous transaction completes.
- Solution would be, instead of running transactions in a serial schedule, we can allow the ***other non-serial schedules which are equivalent to serial***, to be executed.

5.4 Serializability

T1	T2
r(a)=90	
a=a -3	
w(a)=87	
r(b)=90	
b=b+3	
w(b)=93	
c	
	r(a)=87
	a=a+2
	w(a)=89
	c

Is this a serial schedule? Yes
Why? T2 transaction starts execution only after T1 is completed.

Initial values of a=90 and b=90

What is the final value of a and b after completion of T1 and T2?

a= 89
b=93

Is this a correct schedule? Yes

5.4 Serializability

T1	T2
r(a) a= 90	
a=a -3	
	r(a) a= 90
	a=a+2
w(a)=87	
r(b)=90	
	w(a)=92
	c
b=b+3	
w(b)=93	
c	

Is this a serial schedule? No
Why? T2 transaction starts execution before T1 is completed.

Initial values of a=90 and b=90

What is the final value of a and b after completion of T1 and T2?

a= 92
b=93

Is this a correct schedule? No. The final answers are not correct.

T2 still reads **a** as 90 since the changes made by T1 has not written yet.

5.4 Serializability

T1	T2
r(a)=90	
a=a -3	
w(a)=87	
	r(a)=87
	a=a+2
	w(a)=89
	c
r(b)=90	
b=b+3	
w(b)=93	
c	

Is this a serial schedule? No
Why? T2 transaction starts execution before T1 is completed.

Initial values of a=90 and b=90

What is the final value of a and b after completion of T1 and T2?

a= 89
b=93

Is this a correct schedule? Yes. The final answers are correct.

5.4 Serializability

Serial, Nonserial, and Conflict-Serializable Schedules

- According to the examples provided in previous 3 slides, we can see that there can be non-serial schedules which give the expected correct result as well as erroneous results.
- We can use the serializability concept to check whether a given schedule is correct or not.
- Definition for serializability=> *A schedule of n transactions is serializable if it is equivalent to some serial schedule of the same n transactions.*
- For two schedules to be equivalent, ***the operations applied on each data item by the schedule, should be applied to that item in both schedules in the same order.***

5.4 Serializability

Serial, Nonserial, and Conflict-Serializable Schedules

- Given two schedules, if the order of conflicting operations are the same in both schedules, the schedules are ***conflict equivalent.***
- If the order of conflicting operations performed in 2 schedules are different, the effect made on the database would be different. Hence, those 2 schedules are not conflict equivalent.

$S_1 = r_1(X); w_2(X);$

$S_2 = w_2(X); r_1(X);$

S_1 and S_2 are not conflict equivalent since the order of conflicting operations are different.

5.4 Serializability

P

T1	T2
r(a)	
a=a -3	
w(a)	
r(b)	
b=b+3	
w(b)	
c	
	r(a)
	a=a+2
	w(a)
	c

Q

T1	T2
r(a)	
a=a -3	
w(a)	
	r(a)
	a=a+2
	w(a)
	c
r(b)	
b=b+3	
w(b)	
c	

- A schedule *S* is serializable, if it is conflict equivalent to a serial schedule *S'*.

Ex:

- Schedule P is a serial schedule.
- Schedule Q performs all the conflicting operations in the same order as schedule P. Therefore, P and Q schedules are conflict equivalent.
- Hence, Q is a serializable schedule.

5.4 Serializability

Testing for Serializability of a Schedule

- We use an algorithm to determine the conflict serializability of a schedule by constructing a ***precedence graph***.
- The algorithm looks at only the read_item and write_item operations in a schedule.
- It is a directed graph G = (N, E) which has of a set of

  nodes = {T1, T2, … , Tn } and a set of directed

  edges = {e1, e2, … , em }.
- The algorithm is explained in the next slide.

5.4 Serializability

Testing for Serializability of a Schedule

- Algorithm:-
  a. For every transaction T_i in schedule S, create a node labeled T_i in the precedence graph.
  b. For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
  c. For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
  d. For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.
  e. The schedule S is serializable if and only if the precedence graph has no cycles.

5.4 Serializability

Constructing the precedence graph

Step-by-step example

$$S : r_1(X), r_1(Y), w_2(X), w_1(X), r_2(Y)$$

Step 1 – The given schedule S has operations from two transactions. Thus, make two nodes corresponding to the two transactions T1 and T2.

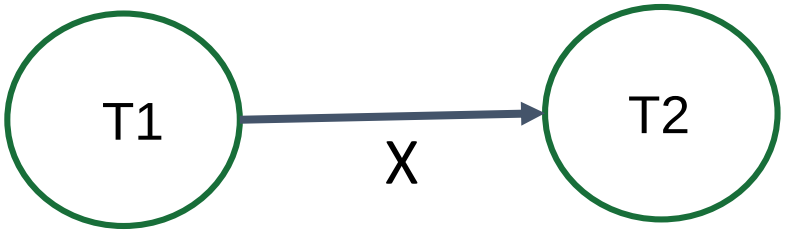
5.4 Serializability

Constructing the precedence graph

Step-by-step example

S : $r_1(X)$, $r_1(Y)$, $w_2(X)$, $w_1(X)$, $r_2(Y)$

Step 2 - For the conflicting pair $r_1(X)$ $w_2(X)$, where $r_1(X)$ happens before $w_2(X)$, draw an edge from T1 to T2.



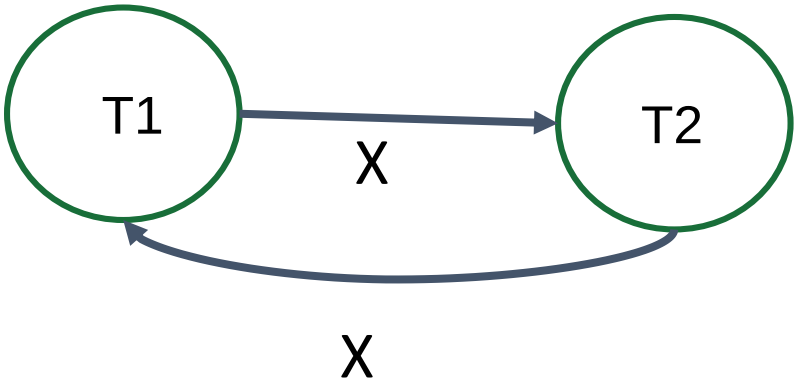
5.4 Serializability

Constructing the precedence graph

Step-by-step example

S : $r_1(X)$, $r_1(Y)$, $w_2(X)$, $w_1(X)$, $r_2(Y)$

Step 3 - For the conflicting pair $w_2(X)$ $w_1(X)$, where $w_2(X)$ happens before $w_1(X)$, draw an edge from T2 to T1.



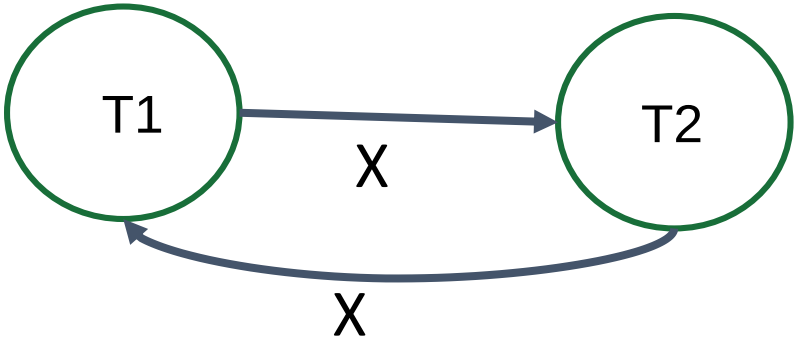
5.4 Serializability

Constructing the precedence graph

Step-by-step example

S : $r_1(X)$, $r_1(Y)$, $w_2(X)$, $w_1(X)$, $r_2(Y)$

Step 3 - Check whether the graph contains cycles.



Since the graph is cyclic, we can conclude that the schedule S is not serializable.

5.4 Serializability

Testing for Serializability of a Schedule

Let's consider another example.

According to the schedule S given in tabular form, we can find the following set of edges for the precedence graph.

Line no. 3 and 4: T2->T3 (Y)

Line no. 1 and 9: T2->T3 (Z)

Line no. 7 and 10: T1->T2 (X)

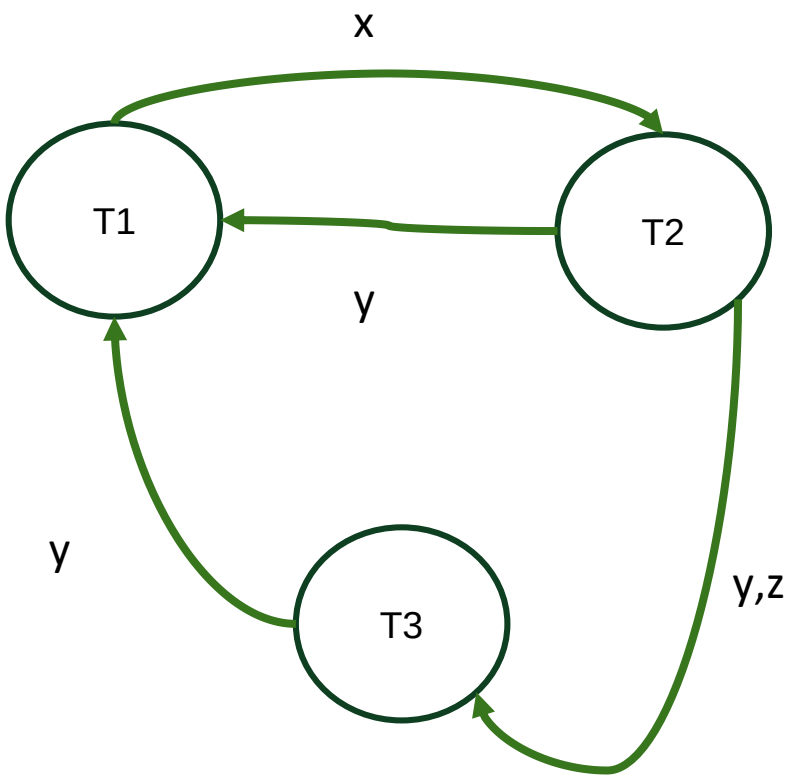
Line no. 3 and 11: T2->T1 (Y)

Line no. 8 and 12: T3->T1 (Y)

	T1	T2	T3
1.		r(Z)	
2		r(Y)	
3		w(Y)	
4			r(Y)
5			r(Z)
6	r(X)		
7	w(X)		
8			w(Y)
9			w(Z)
10		r(X)	
11	r(Y)		
12	w(Y)		
13		r(X)	

5.4 Serializability

Testing for Serializability of a Schedule



Based on the edges identified, we can draw the given graph.

Graph contains cycles.

Therefore, no equivalent serial schedule to the given schedule S exists.

Hence, the schedule S, is not serializable.

5.4 Serializability

Using Serializability for Concurrency Control

- A serial schedule may slow down the execution process, as it does not utilize the processing time efficiently when
  - executing long transactions in a serial schedule
  - waiting for I/O operations
- However, serializable schedules allow concurrent execution without giving up the accuracy.
- In practical scenarios, it is difficult to test serializability of schedules as the execution of processes are determined by the operating system scheduler.
- It is difficult to pre determine the order of operations in advance to ensure serializability.

5.4 Serializability

Using Serializability for Concurrency Control

- Most of the DBMSs have designed a rule set, which are to be followed by all the transactions hence, the result will be a serializable schedule.
- Rarely some may allow non serializable schedules to be executed in order to reduce the overhead of transactions.

5.4 Serializability

View Equivalence and View Serializability

- The idea behind view equivalence is, we need to get the same result from the write operations of transactions as long as the result that is being read by each read operation is generated by the same write operation in both schedules.
- Offers less restrictive definition of schedule equivalence than conflict serializability.
- To be view serializable, a schedule has to be view equivalent to a serial schedule.

5.4 Serializability

View Equivalence and View Serializability

- Criteria for two schedules S and S′ to be view equivalent is as follows.

1. The same set of transactions participate in S & S′, and S & S′ include the same operations of those transactions.
   Simply, S and S' should have same transactions and operations.
2. For any operation $r_i(X)$ of T_i in S, if the value of X read by the operation has been written by an operation $w_j(X)$ of T_j (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation $r_i(X)$ of T_i in S′.

   Simply, S and S' both schedules should read data items from the same source operation

5.4 Serializability

View Equivalence and View Serializability

3. If the operation $w_k(Y)$ of T_k is the last operation to write item Y in S, then $w_k(Y)$ of T_k must also be the last operation to write item Y in S′.

   Simply, the transaction which has done the last write of a particular data item should be same in both S and S' schedules.

5.4 Serializability

View Equivalence and View Serializability

Consider the schedules A and B given to illustrate view serializability.

T1	T2
r(a)	
w(a)	
	r(a)
	w(a)
r(b)	
w(b)	
	r(b)
	w(b)

S

T1	T2
r(a)	
w(a)	
r(b)	
w(b)	
	r(a)
	w(a)
	r(b)
	w(b)

P

5.4 Serializability

View Equivalence and View Serializability

- In both schedules S and P, we can see two transactions T1 and T2. All the operations included in both T1 and T2 are also the same.
- Therefore, the set of transactions and their operations are the same in S and P.
- For every data item, T2 reads what T1 has written in schedule S. In schedule P also we can see the same sequence of T2 reading what T1 has written.
- In S schedule, last write of all the data items is performed by T2. Similarly, in P schedule also last write of data items has done by T2.
- Since all 3 properties are satisfied, we can conclude S and P are view equivalent.
- P is a serial schedule. There for S is a view serializable schedule.

5.5 Transaction Support in SQL

- The basic interpretation of a SQL transaction is same as the already defined concept of a transaction. That is, a transaction is ***a logical unit*** and it is assured to be atomic.
- Consistently, a particular SQL statement is atomic - either it completes the execution without any error, or it fails and the database remains unchanged.

5.5 Transaction Support in SQL

- In SQL, there isn't any explicit "Begin_Transaction" statement. When specific SQL statements are encountered, transaction initiation is done implicitly.
- Every transaction must have an explicit end statement, which is either a "COMMIT" or a "ROLLBACK".
- Further, every transaction is characterized by some attributes. those are;
  - Access mode
  - Diagnostic area size
  - Isolation level
- In SQL, there is "SET TRANSACTION" statement to specify above characteristics

5.5 Transaction Support in SQL

- The **access mode** can be specified as READ ONLY or READ WRITE. The default is READ WRITE.
- READ WRITE allows select, update, insert, delete, and create commands to be executed.
- READ ONLY is simply for data retrieval.
- Syntax → `SET TRANSACTION READ ONLY ;`

  `SET TRANSACTION READ WRITE ;`

5.5 Transaction Support in SQL

- **DIAGNOSTIC SIZE *n***, is the SQL option to set diagnostics area size. ***n*** is an integer value which specifies the number of conditions allowed to have together in the diagnostic area. Also it provides feedback information such as errors and exceptions, about most recently executed ***n*** SQL statements***.***

5.5 Transaction Support in SQL

```
SET TRANSACTION
    READ ONLY,
    ISOLATION LEVEL READ UNCOMMITTED,
    DIAGNOSTIC SIZE 6;
```

This statement defines a transaction which has read only access mode , read uncommitted isolation level and provides feedback information about most recently executed 6 statements.

5.5 Transaction Support in SQL

- **ISOLATION LEVEL <isolation>,** is the SQL option to set isolation level of the transaction. for **<isolation>**, following values can be applied.
    - READ UNCOMMITTED
    - READ COMMITTED
    - REPEATABLE READ
    - SERIALIZABLE (default isolation level)
  - In here, the term “SERIALIZABLE” has been used based on the prevention of violations that produce dirty read, unrepeatable read, and phantoms. (discussed in 5.1.2).
  - Thus, even if the transactions are executed concurrently, serializable isolation makes sure that the outcome of this concurrent execution would produce the same effects as if they were executed serially.

5.5 Transaction Support in SQL

Possible Violations Based on Isolation Levels as Defined in SQL.

	Dirty Read	Non-repeatable read	phantoms
Read Uncommitted	✓	✓	✓
Read Committed	✗	✓	✓
Repeatable Read	✗	✗	✓
Serializable	✗	✗	✗

5.5 Transaction Support in SQL

Read Uncommitted: Declares that transaction can read rows that have been modified by other transactions but not yet committed. Thus, result in dirty reads, non-repeatable reads and phantoms.

- **Example -** Consider the following transactions T1 and T2 occurs on an account that holds Rs.50,000 of initial balance.

  Transaction (T1) →

  Deduct Rs: 1000 from an account (Customer_ID=Cid_1105) due to an automated bill payment happens every month.

  But, since an error occurred, **T1 transaction rolled back** without committing.

  Transaction (T2) →

  At the same time while T1 executes, customer (Customer_ID=Cid_1105) checks his account balance.

5.5 Transaction Support in SQL

- This is the Tabular representation of the Transactions T1 and T2 explained in the example of the previous slide.

T1	T2
Read (balance) balance=balance - 1000	
	Read (balance)
Rollback	

5.5 Transaction Support in SQL

Suppose we have set isolation level to **Read Uncommitted** in T2, as shown in following SQL query.

```
SET TRANSACTION ISOLATION LEVEL READ
UNCOMMITTED;
BEGIN TRAN;
SELECT balance
FROM Customer_tbl
WHERE Customer_ID = 'Cid_1105';
COMMIT TRAN;
```

5.5 Transaction Support in SQL

- We get **`output = 49,000`** as the result of the transaction T2.
- However, the actual balance should be 50,000 since T1 is rolled back to the original value.
- **Explanation**→ 49,000 is the balance updated by T1. T2 reads the balance (as 49,000) before T1 rollback. This **dirty read** occurred because we have set the isolation level to "**`READ UNCOMMITTED`**" in T2.

5.5 Transaction Support in SQL

Read Committed: Declares that transaction can only read data that has been **committed** by other transactions. Thus, prevent dirty reads. But result in non-repeatable reads and phantoms.

- **Example -** Consider the following transactions T1 and T2 occurs on an account that holds Rs.50,000 of initial balance.

  Transaction (T1) →

  Deduct Rs: 1000 from an account (Customer_ID=Cid_1105) due to an automated bill payment happens every month.

  This transaction was successfully completed and committed to the database.

  Transaction (T2) →

  At the same time while T1 executes, customer (Customer_ID=Cid_1105) checks his account balance twice consequently. T2 Reads the account balance twice.

5.5 Transaction Support in SQL

- This is the tabular representation of Transactions T1 and T2 explained in the example of the previous slide.

T1	T2
Read (balance) balance=balance - 1000	
	Read (balance)
Commit	
	Read (balance)

5.5 Transaction Support in SQL

Suppose we have set isolation level to **Read Committed** in T2, as shown in following SQL query.

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN TRAN;
SELECT balance
FROM Customer_tbl
WHERE Customer_ID = 'Cid_1105';
COMMIT TRAN;
```

5.5 Transaction Support in SQL

- We get **`output = 50,000`** as the result of the first read in transaction T2 and **`output = 49,000`** for the second read in transaction T2.
- **Explanation**→ Because we have set the isolation level to "**`READ COMMITTED`**" in T2, it only reads committed data by other transactions. Since T1 is not committed at the first time T2 reads the balance, T2 reads the balance as 50,000.
- Thus will result in non-repeatable read and phantoms.

5.5 Transaction Support in SQL

Repeatable read: Declares that,

- Statements cannot read data that has been modified but not yet committed by other transactions

  and

- No other transactions can modify the data that the current transaction has read until the current transaction has completed.

5.5 Transaction Support in SQL

- **Example -** Consider the transactions T1 and T2 occurs on an account that holds Rs.50,000 of initial balance.

  Transaction (T1) →

  Deduct Rs: 1000 from an account (Customer_ID=Cid_1105) due to an automated bill payment happens every month.

  Then T1 transaction commits.

  Transaction (T2) →

  At the same time while T1 executes, customer (Customer_ID=Cid_1105) checks his account balance twice consequently.

5.5 Transaction Support in SQL

- This is the tabular representation of transactions T1 and T2 explained in the example of the previous slide.

T1	T2
Read (balance) balance=balance - 1000	
	Read (balance)
Commit	
	Read (balance)

5.5 Transaction Support in SQL

Suppose we have set isolation level to **Repeatable Read** in T1, as shown in following SQL query.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRAN;
UPDATE Customer_tbl
SET balance=balance-1000
WHERE Customer_ID = 'Cid_1105';
COMMIT TRAN;
```

5.5 Transaction Support in SQL

- First read statement of T2 will not get the balance, but the second read statement in T2 will get the **`output = 49,000.`**
- **Explanation**→ We have set the isolation level to "**`REPEATABLE READ`**" in T1, the first read statement in T2 will not allowed to read the balance because T1 has updated the balance and not committed yet.
- When T2 reads the balance again, T1 has been completed and committed to the database. Hence it gets the **`output= 49,000.`**

5.5 Transaction Support in SQL

- **Serializable:** Declares that,
  - Statements cannot read data that has been modified but not yet committed by other transactions

    and
  - No other transactions can modify the data that the current transaction has read until the current transaction has completed.
  - Until the current transaction completes, other transactions cannot insert new rows with key values that fall inside the range of keys read by any statements in the current transaction.

5.5 Transaction Support in SQL

- **Example -** Consider the transactions T1 and T2 occurs on an account that holds Rs.50,000 of initial balance.

Transaction (T1) →

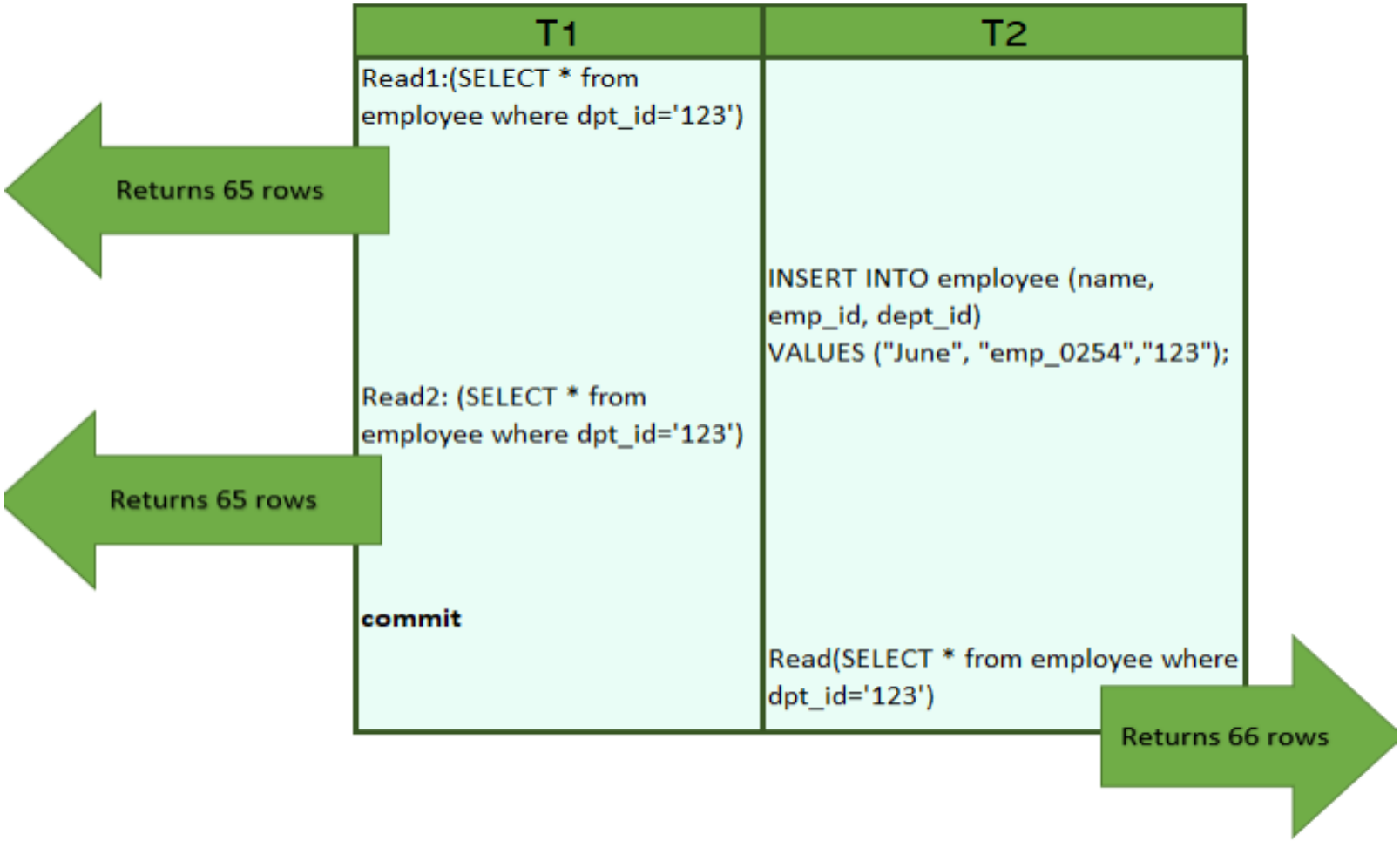
Reads details of employees who are working in the “123” department twice consecutively.

Transaction (T2) →

At the same time new record is inserted into the employee table with name =”June” who is working in the "123" department.

5.5 Transaction Support in SQL

- This is the tabular representation of transactions T1 and T2 explained in the example of the previous slide.



5.5 Transaction Support in SQL

Suppose we have set isolation level to **Serializable** in T1, as shown in following SQL query.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN;
SELECT *
FROM employee
WHERE dept_id=”123”
COMMIT TRAN;
```

5.5 Transaction Support in SQL

- For T1, both read statements will return 65 rows.
- **Explanation**→ We have set the isolation level to "**SERIALIZABLE**" in T1. Therefore T2 cannot insert details of an employee who is working in the"123" department which is the key read by T1.
- When T1 has been completed and committed to the database, T2 will be executed and update the employee table. Then the read in T2 will return 66 rows with new insertion.

5.6 Consistency in NoSQL

Consistency

- As we discussed in previous slides, a transaction leads the database from one consistent state to another.
- In other words, transactions must affect database only in valid ways.

Consistency in NoSQL

- In NoSQL databases, eventual consistency is preferred over immediate consistency.This will be discussed in detail later.

5.6 Consistency in NoSQL

Update Consistency

- Update consistency in NoSQL make sure that write-write conflicts doesn't occur.
- Write-write conflict occurs when two transactions update same data item at the same time. If the server serialize the updates, a lost update occurs.
- There are 2 types of approaches for maintaining consistency.
  - **Pessimistic approach**: Prevents conflicts from occurring.
  - **Optimistic approach**: Let the conflicts occur but detects and takes action to sort them out.

5.6 Consistency in NoSQL

Update Consistency cont.

- Selection of a consistency approach would depend upon the fundamental tradeoff between **safety** (avoiding errors such as update conflicts) and **liveliness** (responding quickly to clients).
- Liveliness is described as “something good will eventually occur” and safety as “something bad does not occur”.
- For example, in a banking system it is important to maintain the consistency of transactions among bank accounts every time. Therefore safety should be the priority.
- In an informational site, which displays real time score of a cricket match, it is important to prioritize liveliness over security.

5.6 Consistency in NoSQL

Update Consistency cont.

- Pessimistic approach (Prevents conflicts from occurring)
  - **Write lock**

    In order to write, a transaction need to acquire a lock on the record. When two transactions attempt to acquire the write lock, system ensures only one transaction can get the lock.

    Second transaction will see the result of first transaction's update before deciding whether to make its own update.

    Pessimistic approaches often severely degrade the responsiveness of a system and may even lead to deadlocks.

5.6 Consistency in NoSQL

Update Consistency cont.

- Optimistic approach (Let the conflicts occur but detects and takes action to sort them out)

  - Conditional update

  Before a transaction updates the value of a data item, it checks whether the value has changed since it's last read. If the value has changed, the update will fail. Update will continue otherwise.

5.6 Consistency in NoSQL

Update Consistency cont.

Conditional update - Example

Samanali *and* ***Krishna*** *read the record* ***A*** *which has the value 100.* ***Samanali*** *wants to add 50 to the* ***A*** *value. Just before writing the value, she checks the value of* ***A,*** *to make sure it has not changed since her last read and then does the modification. Meanwhile* ***Krishna*** *wants to subtract 20 from the value* ***A.*** *Just before the modification, he also checks the value of* ***A*** *to make sure the value remain unchanged as 100. But as* ***Samanali*** *has changed* ***A*** *to 150,* ***Krishna*** *fails to do the update.*

5.6 Consistency in NoSQL

Update Consistency cont.

- Optimistic approach (Let the conflicts occur but detects and takes action to sort them out)

  - **Save both updates and mark as conflicts**

  Allow different modification on the same data item to be completed and then merge those updates. Merging can be done by either showing all the modified values to user and ask him to sort it out, or automatically merged by the system.

5.6 Consistency in NoSQL

Update Consistency cont.

Save both updates and mark as conflicts - Example

Samanali *and* ***Krishna*** *read the record* ***A*** *which has the value 100. Then* ***Samanali*** *add 50 to this value and write it. Meanwhile* ***Krishna*** *subtract 20 from value of* ***A*** *and write it. DBMS will save the both values* ***150*** *(changed by* ***Samanali****) and* ***80*** *(changed by* ***Krishna****) as possible values for* ***A*** *and then mark them as* ***conflicts***.

5.6 Consistency in NoSQL

Read Consistency

- Read consistency in NoSQL will guarantee that readers will always get consistent responses to their requests.
- Read consistency will prevent “inconsistent read” or “read-write conflict”.
- Read consistency will preserve ,
  - **Logical consistency** (ensure that different data items make sense together).
  - **Replication consistency** (ensure that same data item has the same value when read from different replicas).
  - **Session consistency** (within user’s session there is *read-your-writes consistency.* It means once you’ve made an update, you are guaranteed to continue seeing that update).

5.6 Consistency in NoSQL

Read Consistency cont.

- *Session consistency* can be maintained using
  - sticky session (session tied to one node) or
  - version stamps (will be discussed later).
- Sometimes nodes may have replication inconsistencies. However, if there are no further updates, eventually all nodes will be updated to the same value. This is known as **eventual consistency**.
- The length of time where the inconsistency present before eventual consistency is know as **inconsistency window**.
- In this inconsistency window some of the data becomes out of date and they are known as **stale data**.

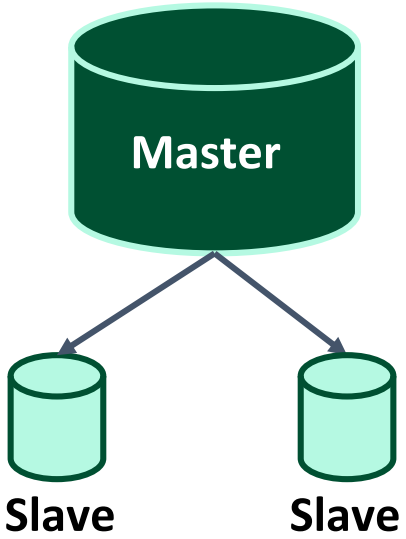
5.6 Consistency in NoSQL

Replication

- Creating multiple copies of data items over different servers is known as replication.
- Can be implemented using following two forms.
  - Master-Slave : In master-slave replication, the master processes the updates and then changes are propagated to slaves.
  - Peer-to-peer: In peer-to-peer replication, all the nodes can process updates and then synchronize their copies of data.

5.6 Consistency in NoSQL

Master-Slave Replication



- Master
  - The authoritative source for the data
  - Responsible for processing updates
  - Can be appointed manually or automatically
- Slaves
  - Changes propagate to slaves from the master
  - If master fails, a slave can be appointed as the new master.

5.6 Consistency in NoSQL

Master-Slave Replication cont.

- Pros

  - Can be easily scale out if more read requests received

  - If master fails, slaves can still handle the read requests

  - Suitable for read-intensive systems

- Cons

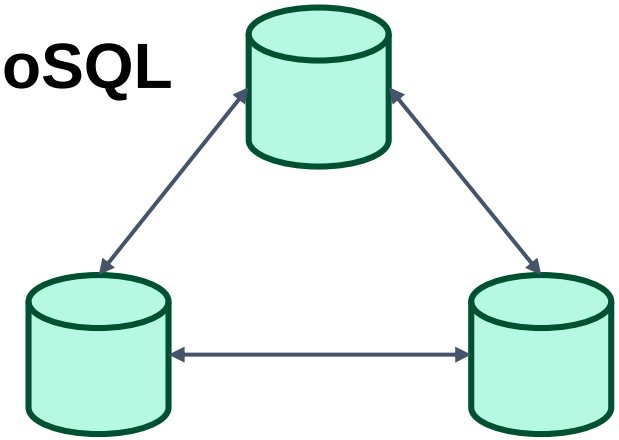
  - Only master can process updates, therefore it may create a bottleneck

  - If master fails, ability to do the updates are eliminated

  - A inconsistency window is inevitable

  - Not suitable for write-intensive systems

5.6 Consistency in NoSQL



Peer-to-Peer Replication

- All the replicas have equal weight
- Every replica can process updates
- Even if one replica fails, system can operate normally.
- Pros
  - Resistant to node failures
  - Can easily add nodes to improve performance
- Cons
  - Write-write inconsistencies can occur
  - Read-write inconsistencies can occur due to slow propagation

5.6 Consistency in NoSQL

Relaxing Consistency

- Even though consistency is a good property, normally it is impossible to achieve consistency without significant sacrifices to other characteristics of the system such as availability.
- Transactions will enforce consistency but it is possible to relax isolation levels to enable individual transactions to read data that has not been committed yet.
- Relaxing isolation level will improve the performance but will reduce the consistency.

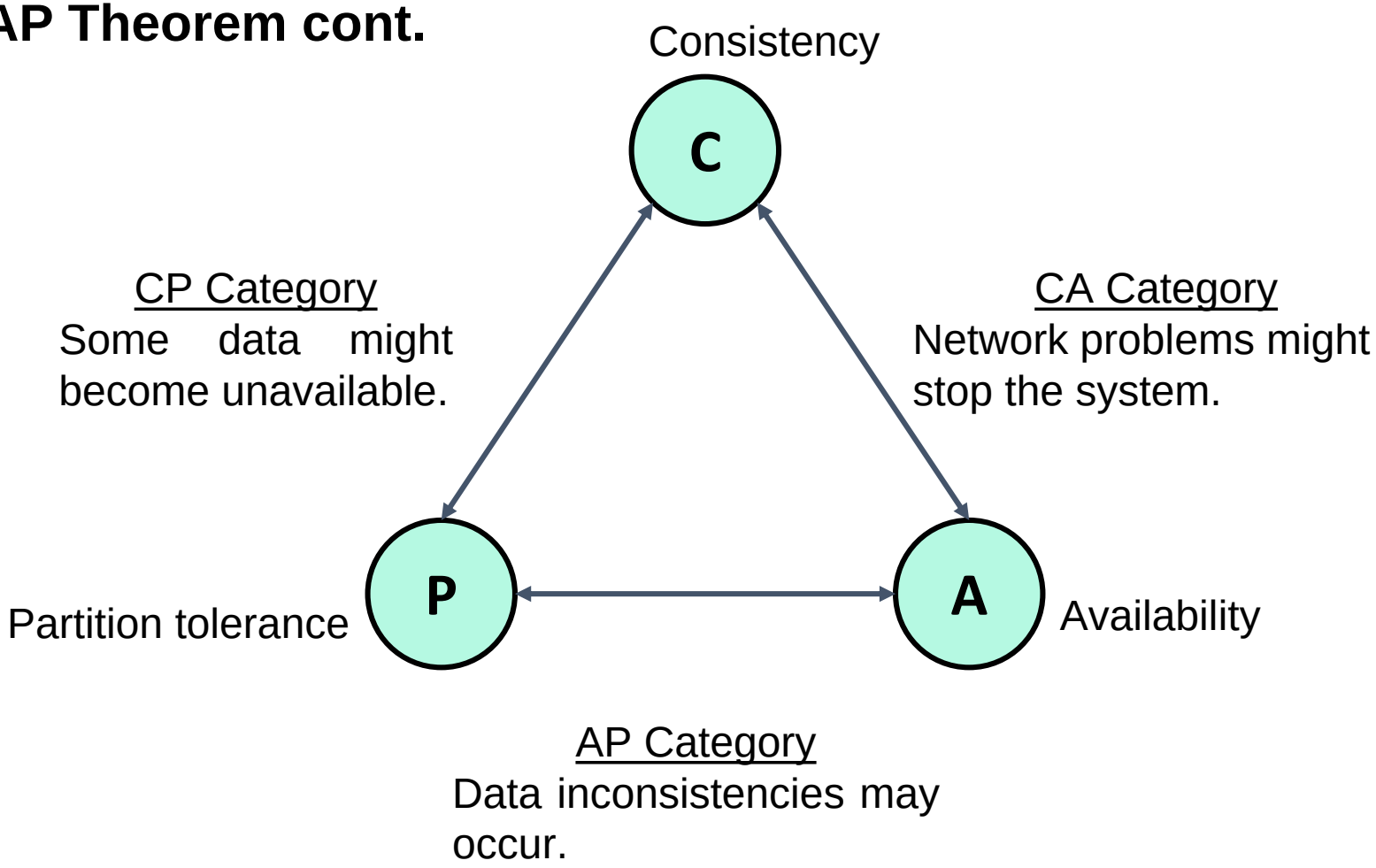
5.6 Consistency in NoSQL

CAP Theorem

- In a database which has several connected nodes, given the three properties of Consistency, Availability and Partition tolerance, it is possible to enforce only two properties at a time.
  - Consistency: (We discussed earlier).
  - Availability: Every request received by a non failing node in the system must result in a response.
  - Partition tolerance: The system continues to operate despite communication breakages that separate the cluster into multiple partitions which are unable to communicate with each other.
- The resulting system designed using CAP theorem will not be perfectly consistent or perfectly available but would have a reasonable combination.

5.6 Consistency in NoSQL

CAP Theorem cont.



5.6 Consistency in NoSQL

CAP Theorem cont.

CAP theorem categorizes systems into three categories.

- CP Category
  - Availability is sacrificed only in the case of a network partition.
- CA Category
  - Consistent and available systems in the absence of any network partition.
- AP Category
  - Systems that are available and partition tolerant but cannot guarantee consistency.

5.6 Consistency in NoSQL

Durability

- Durability means that committed transactions will survive permanently (even if the system crashed). This is achieved by flushing the records to disk (Non-volatile memory) before acknowledging the commit.

Relaxing Durability

- In relaxing durability, database can apply updates in memory and periodically flush changes to the disk. If the durability needs can be specified on a call-by-call basis, more important updates can be flushed to disk.
- By relaxing durability, we can gain higher performance.

5.6 Consistency in NoSQL

Relaxing Durability cont.

- Durability trade off for higher performance may be worthwhile for some scenarios as given below:
  - Storing user-session: There are many activities with respect to a user session, which affect the responsiveness of the website. Thus, losing the session data will create less annoyance than a slower website.

  - Capturing telemetric data from physical devices: It may be necessary to capture data at a faster rate, at the cost of missing the last updates if the server crashes.

5.6 Consistency in NoSQL

Relaxing Durability

- Another class of durability tradeoffs comes up with replicated data.
- A failure of replication durability occurs when a node processes an update but fails before that update is replicated to the other nodes.
- For example, assume a peer-to-peer replicated system with three nodes, R1 , R2 and R3. If the transaction is updated to the memory of R1, but it crashed before the update is sent to R2 and R3, a failure of replication will occurr. This can be avoided by setting the durability level. If the system doesn't acknowledge the commit until the update is propagated to majority of nodes, above scenario will not have occurred.

5.6 Consistency in NoSQL

Quorums

- Answers the question, “How many nodes need to be involved to get strong consistency?”
- Write quorum specifies the number of nodes with non conflicting writes.
- If W > N/2 ; then the system said to have a strong consistency.
  - W - Number of nodes participating in the write
  - N - Number of nodes involved in replication
- The number of replicas is known as the **replication factor.**
- If number of nodes required to contact for a read is R; when R + W > N you can have a strong consistent read.

5.6 Consistency in NoSQL

Quorums Example

- Let's consider a system with replication factor 3. How many nodes are required to confirm a write?

For a system to have a strong consistency, W should be greater than N/2. (N is replication factor)

Here, W needs to be greater than 3/2

W>1.5

Therefore we need at least 2 nodes to confirm a write.

- What is the number of nodes you need to contact for a read?

R + W >N (according to definition in previous slide)

R > N -W

R> 3 - 2

R > 1

Therefore the number of nodes you need to contact for a read is 2.

5.6 Consistency in NoSQL

Version Stamps

- We need human intervention to work with updates in a transactional system as transactions have limitations.
- Applying locks for longer period of time will affect the performance of the system. Solution for this is ***version stamps***, a field that changes every time the underlying data in the record changes.
- System can note the version stamp when reading the data and can check whether it's changed before writing the data.

5.6 Consistency in NoSQL

Version Stamps Cont.

- Version stamps can be created by:
  i. Using an incrementing counter at each update of the resource.
  ii. Create a GUID, which is a large random number that is unique.
  iii. Make a hash of the contents of the resource.
  iv. Use the timestamp of the last update.

We will discuss the advantages and disadvantages of each method in coming slides.

5.6 Consistency in NoSQL

Version Stamps cont.

i. Using an incrementing counter at each update of the resource.

- Pros
  - Easy to compare and find the most recent version
- Cons
  - Requires a server to generate counter values
  - Need a single master to ensure the counters are not duplicated

5.6 Consistency in NoSQL

Version Stamps cont.

ii. Create a GUID

- Pros
  - Can be generated by any node
- Cons
  - Large numbers
  - Unable to compare and find the most recent version directly.

5.6 Consistency in NoSQL

Version Stamps cont.

iii. Make a hash of the content

- Pros
  - Can be generated by any node
  - Deterministic (any node will generate the same hash for the same content)
- Cons
  - Lengthy
  - Cannot be directly compared for recentness

5.6 Consistency in NoSQL

Version Stamps cont.

iv. Use the timestamp of the last update

- Pros
  - Reasonably short
  - Can be directly compared for recentness
  - Does not need single master
- Cons
  - Clocks of all nodes should be synchronized
  - Duplicates can occur if the timestamp is too granular

5.6 Consistency in NoSQL

Version Stamps cont.

- Vector stamps - A special form of version stamp, that is used by peer-to-peer NoSQL systems.
- A vector stamp is a set of counters that are defined for each node.

Example

- Assume there are 3 nodes, A, B and C.
- Vector stamp for these nodes may look like *[A:10,B:15,C:5]*
- When there is an internal update, the node will update its counter.
- Therefore, an update in B will change the vector stamp to *[A:10,B:16,C:5]* (increment B count)
- Whenever two nodes communicates, they synchronize their vector stamps.

Activity

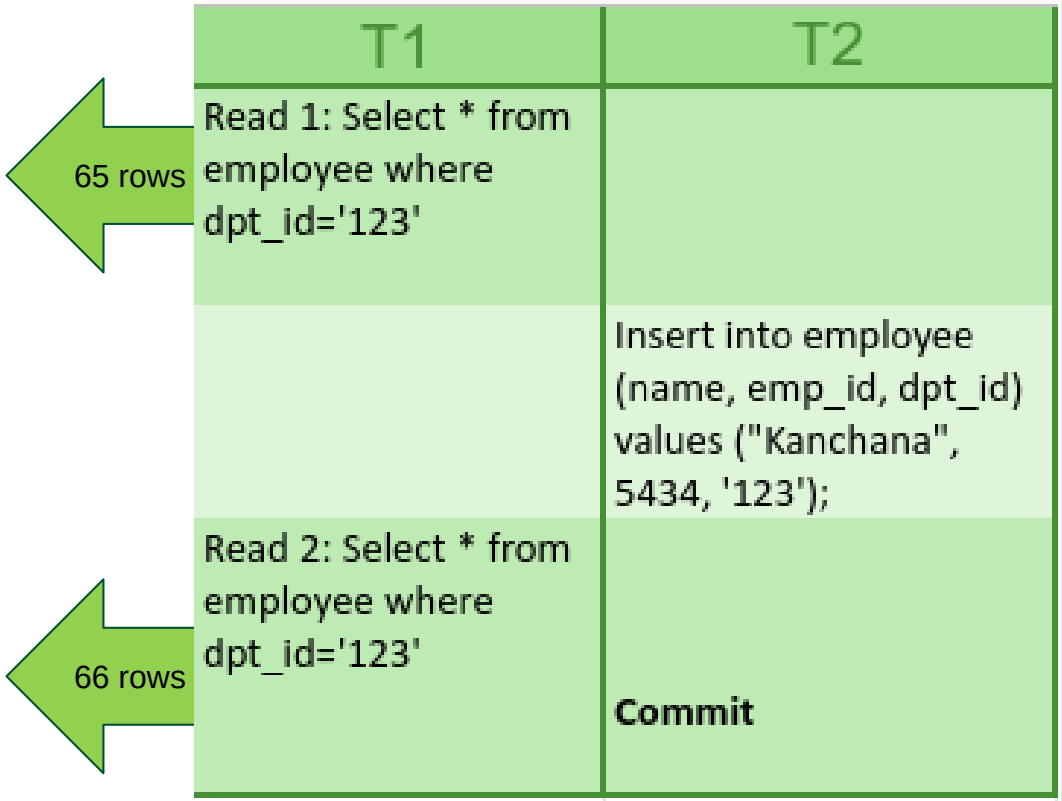
- Check whether the given schedule is serializable by drawing a precedence graph. Justify your answer.

S=r3(Y);r3(Z);r1(X);w1(X);w3(Y);w3(Z);r2(Z);r1(Y);w1(Y);r2(Y);w2(Y);r2(X);w2(X);

Activity

Consider T1 and T2 transactions given in tabular format. If T1 reads 65 row and 66 rows respectively in Read1 and Read2 operations,

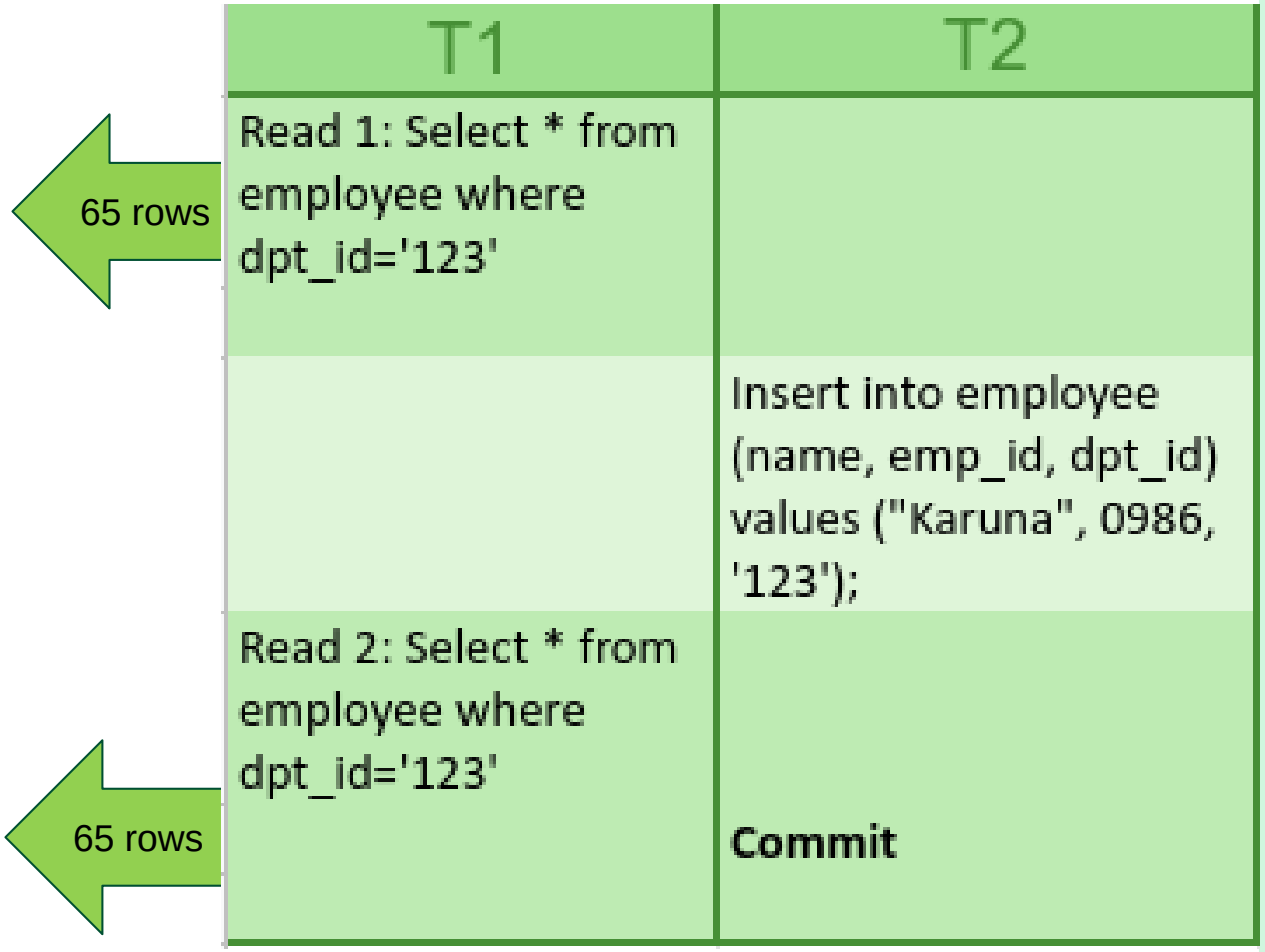
what is the minimum isolation level of transaction T1?



Activity

Consider T1 and T2 transactions given in tabular format. If T1 reads 65 rows in both Read1 and Read2 operations,

what is the minimum isolation level of transaction T1?



Activity

Consider T1 and T2 transactions given in tabular format. If the transaction T1 was executed after setting isolation level as follows,

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRAN;
INSERT INTO employee (name, emp_id, dept_id)
VALUES ("Gabi", 0986,"123");
COMMIT TRAN;
```

What would be the output of transaction T2?

T1	T2
Insert into employee (name, emp_id, dpt_id) values ("Gabi", 0986, '123');	
	Read: Select * from employee where dpt_id='123'
Commit	

Activity

Consider T1 and T2 transactions given in tabular format. Suppose each of the Read operations given in T2 transaction was executed after setting the isolation level as follows.

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN TRAN;
SELECT balance
FROM customer
WHERE customer_ID = '5467';
COMMIT TRAN;
```

T1	T2
	select balance from customer where customer_ID = '5467'
update customer SET balance = balance -1000 WHERE customer_ID = '5467';	
Commit	
	select balance from customer where customer_ID = '5467'

If the customer_ID '5467' had 100,000 in the account at the beginning of T2. What would be the output of second read statement in transaction T2?

Activity

Consider a schedule S with two transactions T1 and T2 as follows;

S: $r_1(X)$; $r_2(X)$; $w_1(Y)$; $w_2(Y)$; c_1; c_2;

Are there conflicting operations in this schedule?

Represent the schedule in a tabular format and explain the conflicting operations if any.

Activity

Consider a schedule S with two transactions T1 and T2 as follows;

S: $r_1(X)$; $w_2(X)$; $r_1(X)$; $w_1(Y)$; c_1; c_2;

Is the schedule S conflict serializable? Provide reasons for you answer.

Activity

Consider the given schedule S for transactions T1, T2 and T3.

S : $r_1(X)$; $r_2(Y)$; $r_3(Z)$; $w_2(Y)$; $w_1(X)$; $w_3(X)$; $r_2(X)$; $w_2(X)$

What is the equivalent serial schedule for the above schedule S ?

Activity

Consider a schedule S with three transactions T1, T2 and T3 as follows.

S: $r_1(X)$; $w_1(X)$; $r_1(Y)$; $r_1(Z)$; c_1; $r_2(X)$; $w_2(X)$; $r_2(Z)$; $w_2(Z)$; c_2; $r_3(Y)$; $w_3(Y)$; $r_3(Z)$; $w_3(Z)$; $c_{3;}$

Is the schedule S a serial schedule? Explain the answer.

Activity

Consider a schedule S with two transactions T1 and T2 as follows.

Is the following schedule S, a recoverable schedule? Justify your answer.

S: $r_1(A)$; $r_2(A)$; $w_1(A)$; $r_1(B)$; $w_2(A)$; $w_1(B)$; c_1; c_2;

Activity

Write whether the given statements regarding schedule S are true or false.

S: $r_1(X)$; $r_2(Y)$; $w_3(X)$; $r_2(X)$; $r_1(Y)$;

1. S is conflict serializable and view serializable. (_____)
2. S does not have any blind writes.(_____)
3. S is conflict serializable but not view serializable. (_____)

Schedule S:

T1	T2	T3	T4
	r(X)		
		w(X)	
		c	
w(X)			
c			
	w(Y)		
	r(Z)		
	c		
			r(X)
			r(Y)
			c

Activity

Write whether the given statements are true or false considering the given schedule S.

1. S is conflict serializable and recoverable. (_____)
2. S is conflict serializable but not recoverable. (_____)
3. S includes blind writes. (_____)
4. S is recoverable but not conflict serializable. (_____)

Activity

Match each of the property given in left column with relevant explanation given in the right column.

Property	Explanation
Consistency	System continues to operate even in the presence of node failure
Availability	System continues to operate in spite of network failures.
Partition Tolerance	All the users can see the same data at same time.

Activity

For a system consisted of 15 nodes with replication factor 5, what is the write quorum and read quorum respectively?

Write quorum =>

Read quorum =>

Activity

Fill in the blanks with the most suitable word provided.

_____ replication is ideal for write intensive system while, _____ replication is better for read intensive system .

In master-slave replication, _____ node is a single point of failure.

Conditional update is a/an _____ approach of maintaining consistency in NoSQL databases,

(peer-to-peer, master-slave, primary, secondary, optimistic, pessimistic)

Activity

- Drag and drop the correct answer from the given list.

Version stamps help users to detect _____ conflicts.

Among the different version stamp creation methods, _____ approach might suffer from getting duplicates if the system get many updates per millisecond.

Version stamp is a field that changes _____, when the underlying data in the record changes.

(concurrency , update , read , version stamp, counters, every time, often, rarely, use the timestamp of the last update, make a hash of the content, create a GUID)

Summary

Introduction to Transaction Processing	Single-user systems, multi-user systems and Transactions Problems in concurrent transaction processing, introduction to concurrency control, DBMS failures, introduction to data recovery Transaction states
Properties of Transactions	ACID properties, levels of isolation

Summary

Schedules	Schedules of Transactions Schedules Based on Recoverability
Serializability	Serial, Nonserial, and Conflict-Serializable Schedules Testing for Serializability of a Schedule Using Serializability for Concurrency Control View Equivalence and View Serializability
Consistency in NoSQL	Update Consistency, Read Consistency, Relaxing Consistency, CAP theorem, Relaxing Durability and Quorums, Version Stamps