USN

Internal Assessment Test III – Aug 2024

Sub:	**Database Management System**			Sub Code:	**BCS/BAD 403**	Branch:	**AINDS / CS (DS)**		
Date:	**07/08/2024**	Duration:	**90 minutes**	Max Marks:	**50**	Sem	**IV**	**OBE**	

		Answer any FIVE Questions	MARKS	CO	RBT
1	a	Describe the database inconsistency problems: Lost update, dirty read and blind write?	5	CO1	L1
	b	With a neat diagram, explain the various states of a transaction execution?	5	CO1	L2
2	a	Check whether the below schedule is conflict or not. {b2 , r2(X) , b1 , r1(X) , wl(X), rl(Y), wl(Y), w2(X), el, c1, e2, c2).	5	CO1	L1
	b	What is 2PL? Explain with an example.	5	CO3	L2
3		How do you detect a deadlock during concurrent transaction execution? Explain Deadlocks and Starvation in 2PL.	10	CO3	L2
4		Explain the various database recovery techniques, with examples.	10	CO3	L3
5		Explain Binary locks and shared locks with algorithms.	10	CO1	L2
6		What is the CAP Theorem? Which of the three properties (consistency, availability, partition tolerance) are most important in NOSQL systems?	10	CO6	L3

CI CCI HOD

USN

Internal Assessment Test III – Aug 2024

Sub:	**Database Management System**			Sub Code:	**BCS/BAD 403**	Branch:	**AINDS / CS (DS)**		
Date:	**07/08/2024**	Duration:	**90 minutes**	Max Marks:	**50**	Sem	**IV**	**OBE**	

		Answer any FIVE Questions	MARKS	CO	RBT
1	a	Describe the database inconsistency problems: Lost update, dirty read and blind write?	5	CO1	L1
	b	With a neat diagram, explain the various states of a transaction execution?	5	CO1	L2
2	a	Check whether the below schedule is conflict or not. {b2 , r2(X) , b1 , r1(X) , wl(X), rl(Y), wl(Y), w2(X), el, c1, e2, c2).	5	CO1	L1
	b	What is 2PL? Explain with an example.	5	CO3	L2
3		How do you detect a deadlock during concurrent transaction execution? Explain Deadlocks and Starvation in 2PL.	10	CO3	L2
4		Explain the various database recovery techniques, with examples.	10	CO3	L3
5		Explain Binary locks and shared locks with algorithms.	10	CO1	L2
6		What is the CAP Theorem? Which of the three properties (consistency, availability, partition tolerance) are most important in NOSQL systems?	10	CO6	L3

CI CCI HOD

USN | | | | | | | | | | |

Internal Assessment Test III –Aug 2024

Sub:	**Database Management System**	Sub Code:	**BCS/BAD 403**	Branch:	**AINDS / CS (DS)**
Date:	**11/07/2024**	Duration:	**90 minutes**	Max Marks:	**50**
Sem	**IV**				**OBE**

		Answer any FIVE Questions	**MARKS**	**CO**	**RBT**
1	A	**1a.Describe the database inconsistency problems: Lost update, dirty read and blind write?**	5	CO1	L1

DBMS has a *Concurrency Control* sub-system to assure database remains in consistent state despite concurrent execution of transactions

Three problems are occurs during concurrency

1. The lost update problem
2. The temporary update (dirty read) problem
3. Incorrect summary problem

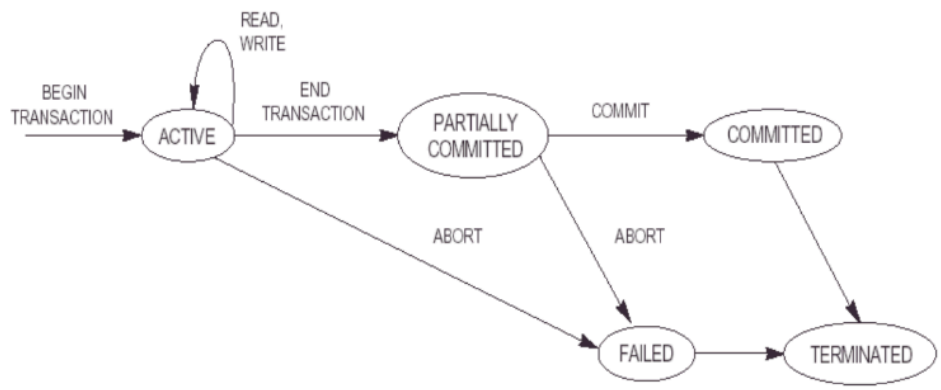
Lost Update Problem

time	T1	T2
	read_item(X);	
	X:=X - N;	
		read_item(X);
		X:=X+M;
	write_item(X);	
	read_item(Y);	
		write_item(X);
	Y:=Y + N;	
	write_item(Y);	

Temporary Update (Dirty Read)

	B	time — T1 — T2 T1: read_item(X); T1: X:=X - N; T1: write_item(X); T2: read_item(X); T2: X:=X+M; T2: write_item(X); T1: read_item(Y); ***T1 fails and aborts*** **Incorrect Summary Problem** time — T1 — T2 T2: sum:=0; T2: read_item(A); T2: sum:=sum+A; T1: read_item(X); T1: X:=X-N; T1: write_item(X); T2: read_item(X); T2: sum:=sum+X; T2: read_item(Y); T1: read_item(Y); T1: Y=Y+N T2: sum:=sum+Y T1: Write_item(Y) **With a neat diagram, explain the various states of a transaction execution?** **Transaction states** • BEGIN_TRANSACTION: marks start of transaction • READ or WRITE: two possible operations on the data • END_TRANSACTION: marks the end of the read or write operations; start checking whether everything went according to plan • COMIT_TRANSACTION: signals successful end of transaction; changes can be “committed” to DB • Partially committed	5	CO1	L2

- ROLLBACK (or ABORT): signals unsuccessful end of transaction, changes applied to DB must be undone



<table>
<tr>
<td>2</td>
<td>a,b</td>
<td>
Check whether the below schedule is conflict or not.

{b2 , r2(X) , b1 , r1(X) , wl(X), rl(Y), wl(Y), w2(X), el, c1, e2, c2).

<table>
<tr><th>T1</th><th>T2</th></tr>
<tr><td></td><td>B2
R2(X)</td></tr>
<tr><td>B1
R1(X)</td><td></td></tr>
<tr><td>W1(X)
R1(Y)
W1(Y)</td><td></td></tr>
<tr><td></td><td>W2(X)</td></tr>
<tr><td>E1
C1</td><td></td></tr>
<tr><td></td><td>E2
C2</td></tr>
</table>
The above schedule is not serializable

Before W2(X) transaction T1 is reading the R1(X) value

The serializable schedule as follows {b2 , r2(X) , b1 , w2(X), r1(X) , wl(X), rl(Y), wl(Y),, el, c1, e2, c2).

What is 2PL? Explain with an example.
</td>
<td>5</td>
<td>CO1</td>
<td>L2</td>
</tr>
</table>

Two-Phase Locking (2PL) Protocol

Transaction is said to follow the *two-phase-locking protocol* if all locking operations precede the *first* unlock operation

- Expanding (growing) phase
- Shrinking phase

During the shrinking phase no new locks can be acquired

- Downgrading ok
- Upgrading is not

2PL Example

T1'	T2'
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
write_lock(X);	write_lock(Y);
unlock(Y);	unlock(X);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

Both T1' and T2' follow the 2PL protocol

Any schedule including T1' and T2' is guaranteed to be serializable

Limits the amount of concurrency

Two-phase locking protocol (2PL)

All lock operations precede the first unlock operation

Expanding phase and shrinking phase

Upgrading of locks must be done in expanding phase and downgrading of locks

must be done in shrinking phase

If every transaction in a schedule follows 2PL protocol then the schedules is guaranteed

to be serializable.

Variants of 2PL

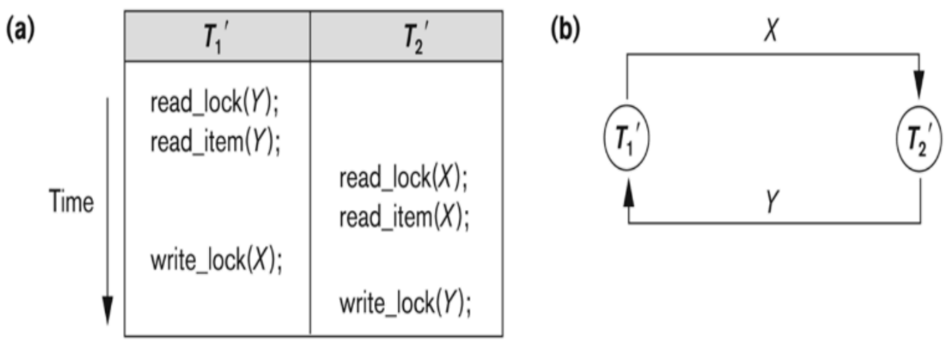
Basic, conservative, strict, and rigorous

How do you detect a deadlock during concurrent transaction execution? Explain Deadlocks and Starvation in 2PL.

2PL can produce deadlocks

Deadlock and starvation in 2PL

Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T' in the set.



Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

Deadlock prevention protocols

- Conservative 2PL, lock all needed items in advance
- Ordering all items in the database

Possible actions if a transaction is involved in a possible deadlock situation

- Block and wait
- Abort and restart
- Preempt and abort another transaction

Two schemes that prevent deadlock (Timestamp based)

Wait-die

An older transaction is allowed to wait on a younger transaction whereas a younger transaction requesting an item from held by an older transaction is aborted and restarted with the same timestamp

Wound-wait

A younger transaction is allowed to wait on an older transaction whereas an older transaction requesting an item from held by a younger transaction preempts the younger transaction by aborting it.

Starvation

A transaction cannot proceed for an infinite period of time while other transactions in the system continue normally

- Unfair waiting scheme
- Victim selection

5	CO2	L3

3		**c)Explain the various database recovery techniques, with examples.**
		
		• Keeps information about operations made by transactions:
		    – *Before-image* (*undo entry*) of updated items
		    – *After-image* (*redo entry*) of updated items
		• Enables restoring a consistent state after non-catastrophic failure (forward/backward).
		• Alternatives:
		    – *undo/no-redo*
		    – *no-undo/redo*
		    – *undo/redo*
		    – *no-undo/no-redo.*
		Write-Ahead Logging (WAL)
		
		(1) No overwrite of disk data before *undo*-type log records are forced to disk.
		
		(2) *Both undo- and redo-type* log records (= before- and after-images) must be forced to disk before end of commit.
		
		Backup
		
		Copy of database on archival storage
		(off-line, often on tape).
		Enables partial recovery from *catastrophic* failures:
		For *committed* transactions:
		Load backup and apply redo operations from the
		log (if the log survived).
		Non-committed transactions must be restarted
		(= re-executed).
		Cache
		
		In-memory buffer for database pages.
		A *directory* (page table) keeps track of pages
		in cache.
		Page-replacement strategy needed, e.g.
		    FIFO (First-In-First-Out), or
		    LRU (Least Recently Used)
		Dirty bit tells for each page, if it has changed
		Flushing means (force-)writing buffer pages to disk.
		Pin/unpin bit tells if the page can be written
		Rollback
		
		At failure, apply undo-type log records (before-images) to updated items.
		A recoverable schedule may allow *cascading rollback*.
		Most practical protocols avoid cascading rollbacks. Then no *read-entries* are required in the log (no dirty reads).
		
		Explain binary locks and shared locks with algorithms?
			10	CO2	L4

Binary locks and shared locks are two important concepts in concurrent programming and database systems. Let's explore each of them:

1. Binary Locks (Exclusive Locks):

Binary locks, also known as exclusive locks or mutex locks, allow only one process or thread to access a shared resource at a time. When a process acquires a binary lock, it has exclusive access to the resource, and all other processes must wait until the lock is released.

Algorithm for Binary Lock:

```
struct BinaryLock {
    bool locked = false;
}

function acquire(BinaryLock lock) {
    while (true) {
        if (!lock.locked) {
            lock.locked = true;
            return;
        }
        // Wait or yield to other processes
    }
}

function release(BinaryLock lock) {
    lock.locked = false;
}
```

2. Shared Locks (Read-Write Locks):

Shared locks allow multiple processes to read a shared resource simultaneously, but ensure exclusive access for writing. This is useful when read operations are more frequent than write operations.

There are two types of locks in this system:

- Read lock: Multiple processes can hold read locks simultaneously.
- Write lock: Only one process can hold a write lock, and no read locks can be held simultaneously.

Algorithm for Shared Lock:

```
struct SharedLock {
    int readers = 0;
    bool writer = false;
```

```
   BinaryLock readLock;
   BinaryLock writeLock;
}

function acquireReadLock(SharedLock lock) {
   acquire(lock.readLock);
   if (lock.readers == 0) {
      acquire(lock.writeLock);
   }
   lock.readers++;
   release(lock.readLock);
}

function releaseReadLock(SharedLock lock) {
   acquire(lock.readLock);
   lock.readers--;
   if (lock.readers == 0) {
      release(lock.writeLock);
   }
   release(lock.readLock);
}

function acquireWriteLock(SharedLock lock) {
   acquire(lock.writeLock);
}

function releaseWriteLock(SharedLock lock) {
   release(lock.writeLock);
}
```

What is the CAP Theorem? Which of the three properties (consistency, availability, partition tolerance) are most important in NOSQL systems?

The CAP Theorem, also known as Brewer's Theorem, is a fundamental concept in distributed computing systems, particularly relevant to distributed databases and NoSQL systems. It states that in a distributed data store, it is impossible to simultaneously guarantee all three of the following properties:

1. Consistency (C): All nodes see the same data at the same time. In other words, a read operation will return the most recent write operation's value, no matter which node it contacts.

2. Availability (A): Every request receives a response, without guarantee that it contains the most recent version of the data. The system remains operational and can respond to requests even if some nodes are down.

3. Partition Tolerance (P): The system continues to operate despite network partitions or communication breakdowns between nodes.

The theorem asserts that a distributed system can only guarantee two out of these three properties at any given time.

In NoSQL systems, which are often designed for large-scale, distributed environments, Partition Tolerance is generally considered non-negotiable. Network partitions are a reality in distributed systems, and the ability to handle them is crucial. Therefore, NoSQL databases typically have to choose between Consistency and Availability when network partitions occur.

Most NoSQL systems prioritize Availability and Partition Tolerance (AP) over strict Consistency. Here's why:

1. Scale: NoSQL databases are often used in scenarios requiring high scalability. Prioritizing availability allows the system to continue functioning even when some nodes are unreachable.

2. Performance: Strict consistency can introduce latency, as the system needs to synchronize all nodes before responding to requests. Many NoSQL use cases prioritize low-latency responses over perfect consistency.

3. Use Case Requirements: Many applications using NoSQL can tolerate eventual consistency, where the system will become consistent over time, rather than requiring immediate consistency.

4. Geographic Distribution: For globally distributed databases, network partitions are more common, and maintaining strict consistency across all regions can be challenging and performance-intensive.

However, it's important to note that this is a generalization. Different NoSQL databases make different trade-offs:

- Cassandra and DynamoDB, for instance, typically favor Availability and Partition Tolerance (AP).
- MongoDB and HBase lean more towards Consistency and Partition Tolerance (CP) in their default configurations.
- Some modern distributed databases like Google's Spanner aim to provide all three properties most of the time, only sacrificing availability during rare network partitions.

The choice between consistency and availability often depends on the specific use case. For example, a financial system might prioritize consistency over availability, while a content delivery system might prioritize availability over strict consistency.

In summary, while Partition Tolerance is crucial for NoSQL systems, the choice between Consistency and Availability is often application-specific. However, many NoSQL systems are designed with a bias towards Availability and Partition Tolerance, with various mechanisms to provide different levels of eventual consistency.

4

Normalize the below relation up to 3NF

Module	Dept	Lecturer	Text
M1	D1	L1	T1
M1	D1	L1	T2
M2	D1	L1	T1
M2	D1	L1	T3
M3	D1	L2	T4
M4	D2	L3	T1
M4	D2	L3	T5
M5	D2	L4	T6

- To normalize the given relation up to Third Normal Form (3NF), we'll follow the normalization process:

Step1:-Identify the functional dependencies.
Step2:-Ensure the relation is in First Normal Form (1NF).
Step3:-Transform it to Second Normal Form (2NF).
Step4:-Transform it to Third Normal Form (3NF)

Step 1: Identify Functional Dependencies

- From the given table:
- **Module** determines **Dept** and **Lecturer**.
- **Module** and **Text** are combined to determine all attributes uniquely, as each combination of **Module** and **Text** uniquely identifies a record.
- Therefore, we have the following functional dependencies:

Module → Dept, Lecturer
Module, Text → Dept, Lecturer, Text

Step 3: Second Normal Form (2NF)

- A table is in 2NF if it is in 1NF and all non-key attributes are fully functionally dependent on the primary key. We need to ensure that there are no partial dependencies on a composite key.
- **Current Candidate Key:** (Module, Text)
- To move to 2NF, we decompose the table to remove partial dependencies.
- **Decomposition into 2NF:**

Table 1: Modules

Module	Dept	Lecturer
M1	D1	L1
M2	D1	L1
M3	D1	L2
M4	D2	L3
M5	D2	L4

Table 2: Module_Text

Module	Text
M1	T1
M1	T2
M2	T1
M2	T3
M3	T4
M4	T1
M4	T5
M5	T6

Step 4: Third Normal Form (3NF)

- A table is in 3NF if it is in 2NF and all the attributes are functionally dependent only on the primary key and there are no transitive dependencies.
- **Table 1: Modules** is already in 3NF since all non-key attributes (Dept, Lecturer) depend only on the primary key (Module).
- **Table 2: Module_Text** is in 3NF since there are no transitive dependencies.

Final Normalized Tables:

Table 1: Modules

Module	Dept	Lecturer
M1	D1	L1
M2	D1	L1
M3	D1	L2
M4	D2	L3
M5	D2	L4

Table 2: Module_Text

Module	Text
M1	T1
M1	T2
M2	T1
M2	T3
M3	T4
M4	T1
M4	T5
M5	T6

These two tables are now in 3NF, ensuring no redundancy and eliminating partial and transitive dependencies.

10 CO1 L2

5		See below	10	CO3	L2

Demonstrate the following constraints in SQL with suitable example:

i) NOT NULL ii) Primary key iii) Foreign key iv) Default v)Unique

i) **NOT NULL :- Ensures that a column cannot have a NULL value**

ii) **Primary key :-** A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table.

- The PRIMARY KEY constraint uniquely identifies each record in a table.
- Primary keys must contain UNIQUE values, and cannot contain NULL values.
- A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns.

SQL PRIMARY KEY on CREATE TABLE

- The following SQL creates a PRIMARY KEY on the "ID" column when the "Persons" table is created:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (ID));
```

SQL PRIMARY KEY on ALTER TABLE

To create a PRIMARY KEY constraint on the "ID" column when the table is already created, use the following SQL:

```
ALTER TABLE Persons
ADD PRIMARY KEY (ID);
```

DROP a PRIMARY KEY Constraint

To drop a PRIMARY KEY constraint, use the following SQL:

```
ALTER TABLE Persons
DROP PRIMARY KEY;
```

iii) Foreign key:- Prevents actions that would destroy links between tables.

- The FOREIGN KEY constraint is used to prevent actions that would destroy links between tables.
- A FOREIGN KEY is a field in one table, that refers to the PRIMARY KEY in another table.
- The following SQL creates a FOREIGN KEY on the "PersonID" column when the "Orders" table is created:

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID));
```

SQL FOREIGN KEY on ALTER TABLE

To create a FOREIGN KEY constraint on the "PersonID" column when the "Orders" table is already created, use the following SQL:

```
ALTER TABLE Orders
ADD FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);
```

DROP a FOREIGN KEY Constraint

To drop a FOREIGN KEY constraint, use the following SQL:

```
ALTER TABLE Orders
DROP FOREIGN KEY FK_PersonOrder;
```

iv) Default:- Sets a default value for a column if no value is specified.

- The **DEFAULT** keyword is used to set a default value for a column. When no value is specified for the column during an INSERT operation, the default value is automatically assigned.

SQL DEFAULT on CREATE TABLE

The following SQL sets a DEFAULT value for the "City" column when the "Persons" table is created:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255) DEFAULT 'Sandnes');
```

SQL DEFAULT on ALTER TABLE

- To create a DEFAULT constraint on the "City" column when the table is already created, use the following SQL:

```
ALTER TABLE Persons
ALTER City SET DEFAULT 'Sandnes';
```

DROP a DEFAULT Constraint

- To drop a DEFAULT constraint, use the following SQL:

```
ALTER TABLE Persons
ALTER City DROP DEFAULT;
```

v)Unique:- Ensures that all values in a column are different.

- The UNIQUE constraint ensures that all values in a column are different.
- Both the UNIQUE and PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns.
- A PRIMARY KEY constraint automatically has a UNIQUE constraint.
- However, many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.

The following SQL creates a UNIQUE constraint on the "ID" column when the "Persons" table is created:

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    UNIQUE (ID));
```

SQL UNIQUE Constraint on ALTER TABLE

- To create a UNIQUE constraint on the "ID" column when the table is already created, use the following SQL:

```
ALTER TABLE Persons
ADD UNIQUE (ID);
```

DROP a UNIQUE Constraint

To drop a UNIQUE constraint, use the following SQL:

<table>
<tr><td></td><td></td><td>ALTER TABLE Persons
DROP INDEX UC_Person;</td><td></td><td></td><td></td></tr>
<tr><td>6</td><td></td><td>Consider the following tables:
works (Pname, Cname, Salary)
lives (Pname, Street, City)
located-In (Cname, City)
write the following queries in SQL:
i) List the names of the people who work for the company 'Wipro' along with the cities they live in.
ii) Find the names of the persons who do not work for 'Infosys'.
iii) Find the people whose salaries are more than that of all of the 'oracle' employees.
iv) Find the persons who works and lives in the same city. (10 Marks)

i) List the names of the people who work for the company 'Wipro' along with the cities they live in.

SELECT w.Pname, l.City FROM works w JOIN lives l ON w.Pname = l.Pname WHERE w.Cname = 'Wipro';

ii) Find the names of the persons who do not work for 'Infosys'.

SELECT l.Pname FROM lives l WHERE l.Pname NOT IN (SELECT w.Pname FROM works w WHERE w.Cname = 'Infosys');

iii) Find the people whose salaries are more than that of all of the 'Oracle' employees.

SELECT w1.Pname FROM works w1 WHERE w1.Salary > ALL (SELECT w2.Salary FROM works w2 WHERE w2.Cname = 'Oracle');

iv) Find the persons who work and live in the same city.

SELECT l.Pname FROM lives l JOIN located_In li ON l.City = li.City JOIN works w ON l.Pname = w.Pname AND w.Cname = li.Cname;</td><td>10</td><td>CO3</td><td>L3</td></tr>
</table>

CI CCI HOD