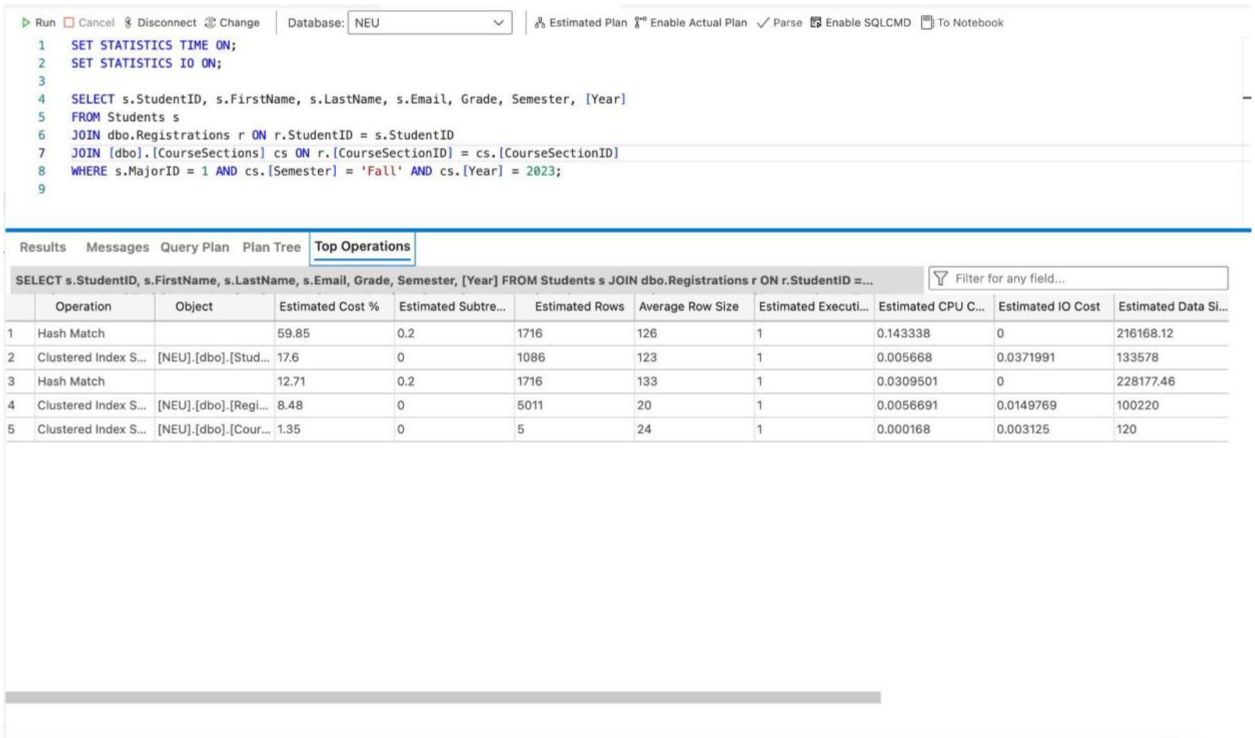
DAMG6210 - Data Management and Database Design

Homework 11

1.



	Operation	Object	Estimated Cost %	Estimated Subtre...	Estimated Rows	Average Row Size	Estimated Executi...	Estimated CPU C...	Estimated IO Cost	Estimated Data Si...
1	Hash Match		59.85	0.2	1716	126	1	0.143338	0	216168.12
2	Clustered Index S...	[NEU].[dbo].[Stud...	17.6	0	1086	123	1	0.005668	0.0371991	133578
3	Hash Match		12.71	0.2	1716	133	1	0.0309501	0	228177.46
4	Clustered Index S...	[NEU].[dbo].[Regi...	8.48	0	5011	20	1	0.0056691	0.0149769	100220
5	Clustered Index S...	[NEU].[dbo].[Cour...	1.35	0	5	24	1	0.000168	0.003125	120

Run Cancel Disconnect Change Database: NEU Estimated Plan Enable Actual Plan Parse Enable SQLCMD To Notebook

```sql
SET STATISTICS TIME ON;
SET STATISTICS IO ON;

SELECT s.StudentID, s.FirstName, s.LastName, s.Email, Grade, Semester, [Year]
FROM Students s
JOIN dbo.Registrations r ON r.StudentID = s.StudentID
JOIN [dbo].[CourseSections] cs ON r.[CourseSectionID] = cs.[CourseSectionID]
WHERE s.MajorID = 1 AND cs.[Semester] = 'Fall' AND cs.[Year] = 2023;
```

Results Messages Query Plan **Plan Tree** Top Operations

SELECT s.StudentID, s.FirstName, s.LastName, s.Email, Grade, Semester, [Year] FROM Students s JOIN dbo.Registrations r ON r.StudentID = s.StudentID JOIN [dbo].[CourseSections] cs ON r....

Operation	Object	Estimated Cost %	Estimated Subtre...	Estimated Rows	Average Row Size	Estimated Executi...
SELECT		0	0.2			
Hash Match		12.71	0.2	1716	133	1
Hash Match		59.85	0.2	1716	126	1
Clustered Index Scan	[NEU].[dbo].[Stud...	17.6	0	1086	123	1
Clustered Index Scan	[NEU].[dbo].[Regi...	8.48	0	5011	20	1
Clustered Index Scan	[NEU].[dbo].[Cour...	1.35	0	5	24	1

Run Cancel Disconnect Change Database: NEU Estimated Plan Enable Actual Plan Parse Enable SQLCMD To Notebook

```sql
SET STATISTICS TIME ON;
SET STATISTICS IO ON;

SELECT s.StudentID, s.FirstName, s.LastName, s.Email, Grade, Semester, [Year]
FROM Students s
JOIN dbo.Registrations r ON r.StudentID = s.StudentID
JOIN [dbo].[CourseSections] cs ON r.[CourseSectionID] = cs.[CourseSectionID]
WHERE s.MajorID = 1 AND cs.[Semester] = 'Fall' AND cs.[Year] = 2023;
```

Results Messages Query Plan **Plan Tree** Top Operations

SELECT s.StudentID, s.FirstName, s.LastName, s.Email, Grade, Semester, [Year] FROM Students s JOIN dbo.Registrations r ON r.StudentID = s.StudentID JOIN [dbo].[CourseSections] cs ON r....

e...	Estimated Rows	Average Row Size	Estimated Executi...	Estimated CPU C...	Estimated IO Cost	Estimated Data Si...	Parallel	Ordered	Estimated Rewinds	Estimated Rebinds
	1716	133	1	0.0309501	0	228177.46	false		0	0
	1716	126	1	0.143338	0	216168.12	false		0	0
	1086	123	1	0.005668	0.0371991	133578	false	false	0	0
	5011	20	1	0.0056691	0.0149769	100220	false	false	0	0
	5	24	1	0.000168	0.003125	120	false	false	0	0

Run Cancel Disconnect Change Database: NEU Estimated Plan Enable Actual Plan Parse Enable SQLCMD To Notebook

```sql
SELECT s.StudentID, s.FirstName, s.LastName, s.Email, Grade, Semester, [Year]
FROM Students s
JOIN dbo.Registrations r ON r.StudentID = s.StudentID
JOIN [dbo].[CourseSections] cs ON r.[CourseSectionID] = cs.[CourseSectionID]
WHERE s.MajorID = 1 AND cs.[Semester] = 'Fall' AND cs.[Year] = 2023;

CREATE NONCLUSTERED INDEX IX_Students_MajorID ON Students(MajorID);
CREATE NONCLUSTERED INDEX IX_CourseSections_Semester_Year ON CourseSections(Semester, [Year]);
CREATE NONCLUSTERED INDEX IX_Registrations_StudentID_CourseSectionID ON Registrations(StudentID, CourseSectionID);
```

Results Messages Query Plan **Plan Tree** Top Operations

SELECT s.StudentID, s.FirstName, s.LastName, s.Email, Grade, Semester, [Year] FROM Students s JOIN dbo.Registrations r ON r.StudentID = s.StudentID JOIN [dbo]....

Operation	Object	Estimated Cost %	Estimated Subtre...	Estimated Rows	Average Row Size
SELECT		0	0.2		
Hash Match		12.71	0.2	1716	133
Hash Match		59.85	0.2	1716	126
Clustered Index Scan	[NEU].[dbo].[Regi...	8.48	0	5011	20
Clustered Index Scan	[NEU].[dbo].[Stud...	17.6	0	1086	123
Index Seek	[NEU].[dbo].[Cour...	1.35	0	5	24

Run Cancel Disconnect Change Database: NEU Estimated Plan Enable Actual Plan Parse Enable SQLCMD To Notebook

```sql
SELECT s.StudentID, s.FirstName, s.LastName, s.Email, Grade, Semester, [Year]
FROM Students s
JOIN dbo.Registrations r ON r.StudentID = s.StudentID
JOIN [dbo].[CourseSections] cs ON r.[CourseSectionID] = cs.[CourseSectionID]
WHERE s.MajorID = 1 AND cs.[Semester] = 'Fall' AND cs.[Year] = 2023;

CREATE NONCLUSTERED INDEX IX_Students_MajorID ON Students(MajorID);
CREATE NONCLUSTERED INDEX IX_CourseSections_Semester_Year ON CourseSections(Semester, [Year]);
CREATE NONCLUSTERED INDEX IX_Registrations_StudentID_CourseSectionID ON Registrations(StudentID, CourseSectionID);
```

Results Messages Query Plan **Plan Tree** Top Operations

SELECT s.StudentID, s.FirstName, s.LastName, s.Email, Grade, Semester, [Year] FROM Students s JOIN dbo.Registrations r ON r.StudentID = s.StudentID JOIN [dbo]....

Average Row Size	Estimated Executi...	Estimated CPU C...	Estimated IO Cost	Estimated Data Si...	Parallel	Ordered	Estimated Rewinds	Estimated Rebinds
133	1	0.0309501	0	228177.46	false		0	0
126	1	0.143338	0	216168.12	false		0	0
20	1	0.0056691	0.0149769	100220	false	false	0	0
123	1	0.005668	0.0371991	133578	false	false	0	0
24	1	0.0001625	0.003125	120	false	true	0	0

Run Cancel Disconnect Change Database: NEU Estimated Plan Enable Actual Plan Parse Enable SQLCMD To Notebook

```sql
SET STATISTICS TIME ON;
SET STATISTICS IO ON;

SELECT s.StudentID, s.FirstName, s.LastName, s.Email, Grade, Semester, [Year]
FROM Students s
JOIN dbo.Registrations r ON r.StudentID = s.StudentID
JOIN [dbo].[CourseSections] cs ON r.[CourseSectionID] = cs.[CourseSectionID]
WHERE s.MajorID = 1 AND cs.[Semester] = 'Fall' AND cs.[Year] = 2023;

CREATE NONCLUSTERED INDEX IX_Students_MajorID ON Students(MajorID);
CREATE NONCLUSTERED INDEX IX_CourseSections_Semester_Year ON CourseSections(Semester, [Year]);
CREATE NONCLUSTERED INDEX IX_Registrations_StudentID_CourseSectionID ON Registrations(StudentID, CourseSectionID);


-- Index on Students table for filtering and joins
CREATE NONCLUSTERED INDEX IX_Students_MajorID_StudentID ON Students(MajorID, StudentID);

-- Drop the existing index
DROP INDEX IX_Students_MajorID_StudentID ON dbo.Students;

```

Results Messages Query Plan **Plan Tree** Top Operations

SELECT s.StudentID, s.FirstName, s.LastName, s.Email, Grade, Semester, [Year] FROM Students s JOIN dbo.Registrations r ON r.StudentID = s.StudentID JOIN [dbo]....

Operation	Object	Estimated Cost %	Estimated Subtre...	Estimated Rows	Average Row Size
SELECT		0	0.2		
Hash Match		12.71	0.2	1716	133
Hash Match		59.85	0.2	1716	126
Clustered Index Scan	[NEU].[dbo].[Regi...	8.48	0	5011	20
Clustered Index Scan	[NEU].[dbo].[Stud...	17.6	0	1086	123
Index Seek	[NEU].[dbo].[Cour...	1.35	0	5	24

Run Cancel Disconnect Change Database: NEU Estimated Plan Enable Actual Plan Parse Enable SQLCMD To Notebook

```sql
-- Index on Students table for filtering and joins
CREATE NONCLUSTERED INDEX IX_Students_MajorID_StudentID ON Students(MajorID, StudentID);

-- Drop the existing index
DROP INDEX IX_Students_MajorID_StudentID ON dbo.Students;


-- Recreate the index with the correct structure
CREATE NONCLUSTERED INDEX IX_Students_MajorID_StudentID ON dbo.Students(MajorID, StudentID);



-- Index on CourseSections table for filtering and joins
CREATE NONCLUSTERED INDEX IX_CourseSections_Semester_Year_CourseSectionID
ON CourseSections(Semester, [Year], CourseSectionID);

```

Results Messages Query Plan **Plan Tree** Top Operations

SELECT s.StudentID, s.FirstName, s.LastName, s.Email, Grade, Semester, [Year] FROM Students s JOIN dbo.Registrations r ON r.StudentID = s.StudentID JOIN [dbo]....

Average Row Size	Estimated Executi...	Estimated CPU C...	Estimated IO Cost	Estimated Data Si...	Parallel	Ordered	Estimated Rewinds	Estimated Rebinds
133	1	0.0309501	0	228177.46	false		0	0
126	1	0.143338	0	216168.12	false		0	0
20	1	0.0056691	0.0149769	100220	false	false	0	0
123	1	0.005668	0.0371991	133578	false	false	0	0
24	1	0.0001625	0.003125	120	false	true	0	0

a) Students Table Index (IX_Students_MajorID):

- Optimizes the WHERE s.MajorID = 1 filter
- Reduces table scanning by directly accessing relevant rows
- Estimated CPU cost: 0.0309501 (relatively low)

b) CourseSections Index (IX_CourseSections_Semester_Year):

- Improves filtering on Semester = 'Fall' AND Year = 2023
- Composite index helps in avoiding table scans
- Estimated IO cost: 0 (very efficient)

c) Registrations Index (IX_Registrations_StudentID_CourseSectionID):

- Optimizes the JOIN conditions
- Covers both join columns for efficient lookup
- Estimated Data Size: 228177.46 (shows data volume handled)

2. The PRIMARY KEY and NOT NULL constraints directly support Consistency because they enforce data integrity rules that ensure the database is in a valid state.

1. PRIMARY KEY on CustomerID:

Ensures that each CustomerID is unique and not null, which guarantees that every customer can be identified by a distinct, valid identifier. This maintains the consistency of the data by preventing duplicate or missing customer IDs.

2. NOT NULL on CustomerName:

Ensures that every customer has a valid name, preventing incomplete or missing data. This supports consistency by enforcing the rule that customer names must always have a value.

Together, these constraints ensure that the data in the Customer table adheres to predefined rules, keeping the database in a consistent and valid state by rejecting any transaction that would violate these rules.

3. The Isolation property of ACID ensures the integrity of data reads.

- When multiple transactions are happening simultaneously, Isolation makes sure that the operations of one transaction do not interfere with the operations of another. It ensures that data being read by one transaction is not affected by others that are still in progress. This prevents situations like dirty reads, where one transaction reads data that is being modified by another, or non-repeatable reads, where the data a transaction reads could change if the transaction reads it again.
- In simple terms, Isolation ensures that each transaction is executed as if it’s the only one happening at that time, keeping the data reads consistent and reliable.

4. Durability ensures that once a transaction is committed, its changes are permanently saved to the database and will not be lost, even if there is a system failure like a crash or power loss.

In this case, if data fails to be written to non-volatile memory (like a hard drive or solid-state drive), it means that the changes made by the transaction are not permanently saved. This violates the Durability property because the transaction's effects would be lost if the system crashes after the transaction was marked as committed.

5. Concurrency control is important in databases because multiple transactions can happen at the same time, and without proper control, this can lead to problems with data accuracy and integrity. Here's why it's needed:

1. Preventing Data Inconsistency
    - When multiple transactions try to read or change the same data at the same time, it can lead to inconsistent data. For example, two transactions might try to update the same record, and one transaction’s changes could overwrite the other’s, causing incorrect data.
2. Ensuring the ACID Properties
    - Concurrency control helps to maintain the ACID properties (Atomicity, Consistency, Isolation, Durability). Without it, transactions might not behave like they should, and this could break rules like Isolation, meaning transactions might interfere with each other and cause incorrect results.
3. Preventing Lost Updates
    - Lost updates happen when two transactions change the same data, but one transaction’s changes get overwritten by the other. This leads to the loss of some updates.
    - For example, two people might try to withdraw money from the same bank account at the same time, but one person's withdrawal gets lost because it was overwritten by the other.
4. Avoiding Dirty Reads
    - A dirty read occurs when a transaction reads data that is being changed by another transaction that hasn’t finished yet. If the second transaction gets rolled back, the first transaction might have used incorrect data.
    - For example, if you read a balance before a transaction is committed, and that transaction gets canceled, your read might have been based on incorrect data.
5. Preventing Non-Repeatable Reads
    - A non-repeatable read happens when you read data, and before you can read it again, another transaction changes it. This can lead to inconsistent results.
    - For example, if you read a product’s price, and then after a few minutes, the price changes because another transaction updated it, you might get different results when you read it again.
6. Avoiding Phantom Reads
    - Phantom reads occur when you query data (like all customers in a certain age range) and another transaction adds, deletes, or modifies records in a way that changes the result of your query.

- For example, if you read all orders from the database, and while you're still working, someone adds new orders that match your query, the data you read might not match the data you expect.

7. Allowing More Transactions to Run Simultaneously
   - Proper concurrency control helps multiple transactions run at the same time while keeping data safe. This makes the system more efficient because transactions don't have to wait for each other to finish, which helps improve performance.
8. Handling Deadlocks
   - Sometimes, transactions can get stuck waiting for each other to release resources, which is called a deadlock. Concurrency control helps manage and resolve deadlocks, so the system doesn't slow down or crash due to these issues.

6.

Feature	Local Transaction	Distributed Transaction
Scope	Involves one database system	Involves multiple databases or systems, possibly across different locations
System Involved	Managed by a single DBMS	Managed by multiple DBMSs, which may be on different machines or platforms
Transaction Manager	Managed by the local DBMS	Requires a global transaction manager to coordinate between multiple systems
Commit Process	The DBMS handles commit or rollback internally	Requires a protocol (e.g., Two-Phase Commit (2PC)) to coordinate commit or rollback across systems
Complexity	Simple, as it's confined to one database.	Complex, as it involves coordination between multiple databases.
Failure Handling	If failure occurs, the transaction is rolled back by the DBMS	Must ensure all databases involved either commit or roll back together. If one system fails, the transaction is usually rolled back across all databases
Transaction Integrity	ACID properties are guaranteed within a single database	ACID properties must be maintained across distributed systems, which adds complexity
Communication	No inter-system communication required	Requires communication between different systems to synchronize the transaction
Example	Transferring funds between accounts in the same bank database	Transferring funds between accounts in two different banks, each using different database systems
Concurrency Control	Single DBMS handles concurrency issues	Concurrency control must be coordinated between multiple systems to avoid conflicts
Recovery from Failure	Easier, as only one system needs to be restored	More complex recovery since all involved systems must be consistent after failure

Performance	Generally, faster as only one DBMS is involved	Can be slower due to network latency and the need for coordination between systems

7. You use the SAVE TRANSACTION statement in SQL when you want to create a savepoint within a transaction. A savepoint is like a checkpoint within your transaction, and it allows you to roll back only part of the transaction, rather than undoing the entire thing.
This is helpful when you're working with complex transactions that involve multiple steps or operations, and you don't want to lose everything if something goes wrong during one part of the transaction.

1. Error Handling:

If you're running a transaction with multiple operations, and something fails (for example, an insert into a table), you can use SAVE TRANSACTION to create a savepoint. If an error occurs later in the transaction, you can roll back to the savepoint instead of rolling back the entire transaction. This helps you keep the changes that were successful.

Example: If you're inserting data into two tables and the second insert fails, you can roll back just the second insert, but keep the first one.

2. Partial Rollback:

Sometimes, you only need to undo part of a transaction. If you set a savepoint before an operation, you can rollback to that savepoint without affecting earlier operations.

Example: Imagine you're updating multiple records in a database. If one update fails, you can roll back to the savepoint set before that update and try again, while keeping the previous successful updates intact.

3. Complex Transactions:

In cases where a transaction involves multiple steps or conditional logic, setting savepoints at different points in the transaction allows you to control errors more precisely and handle failures more effectively.

4. Simulating Nested Transactions:

SQL doesn't support nested transactions, but you can use SAVE TRANSACTION to simulate nested transactions. This helps you logically break down your larger transaction into smaller, more manageable parts.

```sql
BEGIN TRANSACTION;

-- Insert data into the first table
INSERT INTO Table1 (Column1, Column2) VALUES ('Value1', 'Value2');

-- Create a savepoint before the next operation
SAVE TRANSACTION SavePoint1;

-- Insert data into the second table
INSERT INTO Table2 (Column1, Column2) VALUES ('Value1', 'Value2');

-- If an error happens, roll back to the savepoint
IF ERROR_OCCURRED
BEGIN
    ROLLBACK TRANSACTION SavePoint1;  -- Rollback to the savepoint, keeping the first insert
END

-- Continue with other operations or commit the transaction
COMMIT TRANSACTION;
```

8.

Aspect	Row-level locking	Page-level locking
Lock Granularity	Locks individual rows in a table.	Locks an entire page (a group of rows stored together, typically 8 KB in size).
Concurrency	Provides high concurrency because only specific rows are locked, allowing other transactions to access rows not involved in the lock.	Provides lower concurrency because the lock affects all rows in the page, restricting access to unrelated rows.
Overhead	Has higher overhead because it requires more locks to be managed, especially for large transactions that access many rows.	Has lower overhead because fewer locks are required as the lock operates at the page level.
Performance Impact	Better for systems with high contention and frequent access to small subsets of data, as other rows remain accessible.	Better for bulk operations like large queries or updates that involve multiple rows on the same page, reducing lock management overhead.
Deadlock Risk	Lower risk of deadlocks because fewer resources are locked at a time.	Higher risk of deadlocks as more rows may be unnecessarily locked, increasing contention.

Use Cases	Ideal for highly concurrent systems where many users need to update or access different rows simultaneously.	Suitable for batch processing, data warehousing, or operations that process multiple rows in the same page.
Lock Scope	Limited to specific rows, minimizing the impact on unrelated data.	Covers all rows in the page, even those not directly involved in the transaction.
Conflict Likelihood	Less likely to block other transactions accessing unrelated rows.	More likely to block other transactions if they need to access rows on the same page.
Example Scenario	A bank application updating the balance of a specific customer account.	A reporting system fetching or updating rows from a single page in a bulk operation.

9. Yes, a user can influence the locking behavior of the database system, and this is often done to optimize performance or control data consistency in specific situations. Most database systems allow users to manage or adjust locking through several techniques, such as lock hints, setting transaction isolation levels, and using explicit locking commands.

Ways Users Can Influence Locking:

1. Using Lock Hints:
   Lock hints are added to SQL queries to control how locks are applied to the data. For example:
   - NOLOCK: Reads data without placing locks, even if it's uncommitted.
   - UPDLOCK: Locks rows for updates to prevent other modifications.
   - HOLDLOCK: Holds shared locks until the transaction finishes

Example:
*SELECT * FROM Customers WITH (NOLOCK);*

2. Setting Transaction Isolation Levels:

Isolation levels determine how transactions interact with each other. By adjusting the isolation level, a user can control which locks are used.

- Read Uncommitted: No locks, allows dirty reads.
- Serializable: Applies strict locks to prevent phantom reads.

Example:
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

3. Explicit Locking Commands:

Users can directly lock rows or tables during a transaction to control access. For instance, in PostgreSQL:

- FOR UPDATE: Explicitly locks rows to prevent others from modifying them

Example:

*SELECT * FROM Orders WHERE OrderID = 123 FOR UPDATE;*

4. In some systems, users can disable or configure lock escalation, which helps manage how locks are applied as transactions grow

References

Hoffer, J. A., Ramesh, V., & Topi, H. (2016). *Modern database management* (13th ed.). Pearson.