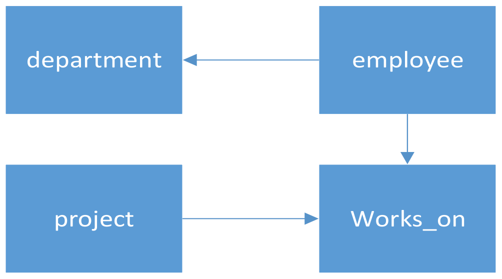
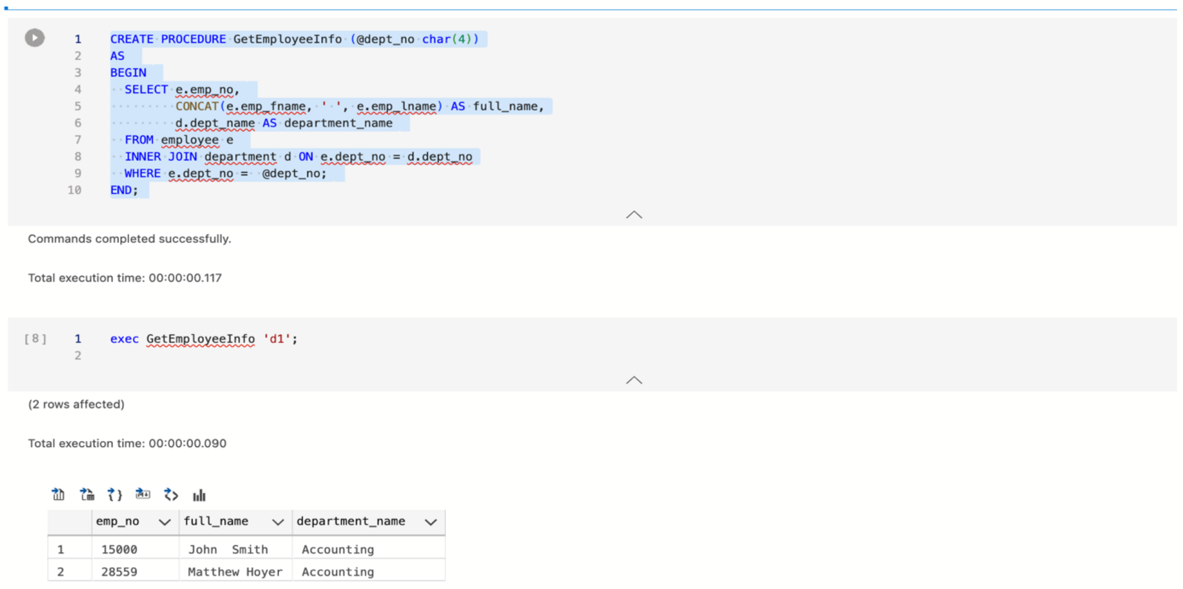
ASSIGNMENT 05

Use the sample database created in previous lecture to answer the following three questions.



1) **Create a stored procedure GetEmployeeInfo which takes @dept_no as an input parameter, and outputs a result set which includes the following fields: emp_no, employee full name, department name. Provide a screenshot of output results using 'd1' as input parameter.**

```
CREATE PROCEDURE GetEmployeeInfo (@dept_no char(4))
AS
BEGIN
 SELECT e.emp_no,
       CONCAT(e.emp_fname, ' ', e.emp_lname) AS full_name,
       d.dept_name AS department_name
 FROM employee e
 INNER JOIN department d ON e.dept_no = d.dept_no
 WHERE e.dept_no =  @dept_no;
END;
```



```
CREATE PROCEDURE GetEmployeeInfo (@dept_no char(4))
AS
BEGIN
  SELECT e.emp_no,
         CONCAT(e.emp_fname, ' ', e.emp_lname) AS full_name,
         d.dept_name AS department_name
  FROM employee e
  INNER JOIN department d ON e.dept_no = d.dept_no
  WHERE e.dept_no =  @dept_no;
END;
```

Commands completed successfully.

Total execution time: 00:00:00.117

```
[8]  exec GetEmployeeInfo 'd1';
```

(2 rows affected)

Total execution time: 00:00:00.090

	emp_no	full_name	department_name
1	15000	John Smith	Accounting
2	28559	Matthew Hoyer	Accounting

2) **Create a stored procedure IncreaseBudgetAmount which takes @project_no and @new_budget as input parameters and returns @message as an output parameter. The stored procedure must perform the following business rules:**

   a. If @project_no is not found, it returns the message “Invalid Project Number”
   b. If @new_ budget is greater than the current budget amount, it must update the project budget and return the message “budget amount increased”
   c. If @new_ budget is less than or equal to the current budget, it does nothing and return the message “New budget must be greater than the current budget”

   Provide sample execution commands for all three business cases, along with a screen shot of results for each.

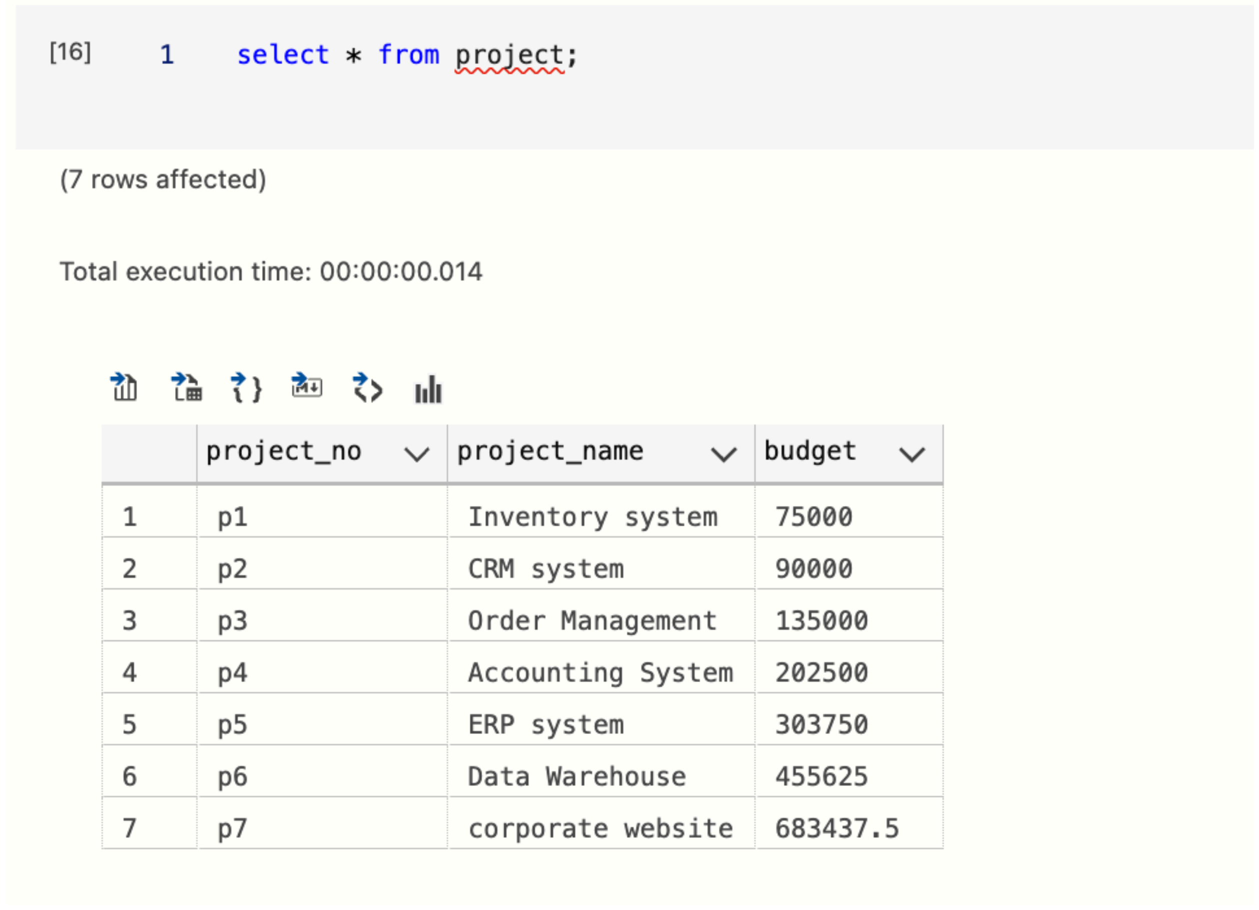
```
ALTER PROCEDURE IncreaseBudgetAmount
  @project_no char(4),
  @new_budget FLOAT,
  @message VARCHAR(100) OUTPUT
AS
BEGIN
  DECLARE @current_budget DECIMAL(10, 2);

  -- if project number exists
  IF NOT EXISTS (SELECT 1 FROM project WHERE project_no = @project_no)
  BEGIN
    SET @message = 'Invalid Project Number';
    RETURN;
  END;

  -- save the current budget
  SELECT @current_budget = budget FROM project WHERE project_no = @project_no;

  -- if new budget is greater than current budget
  IF @new_budget > @current_budget
  BEGIN
    UPDATE project SET budget = @new_budget WHERE project_no = @project_no;
    SET @message = 'Budget amount increased';
  END
  ELSE
  --if new budget is less than current budget
  BEGIN
    SET @message = 'New budget must be greater than the current budget';
  END;
END;
```

Project table state before executing the procedure



```
[16]    select * from project;
```

(7 rows affected)

Total execution time: 00:00:00.014

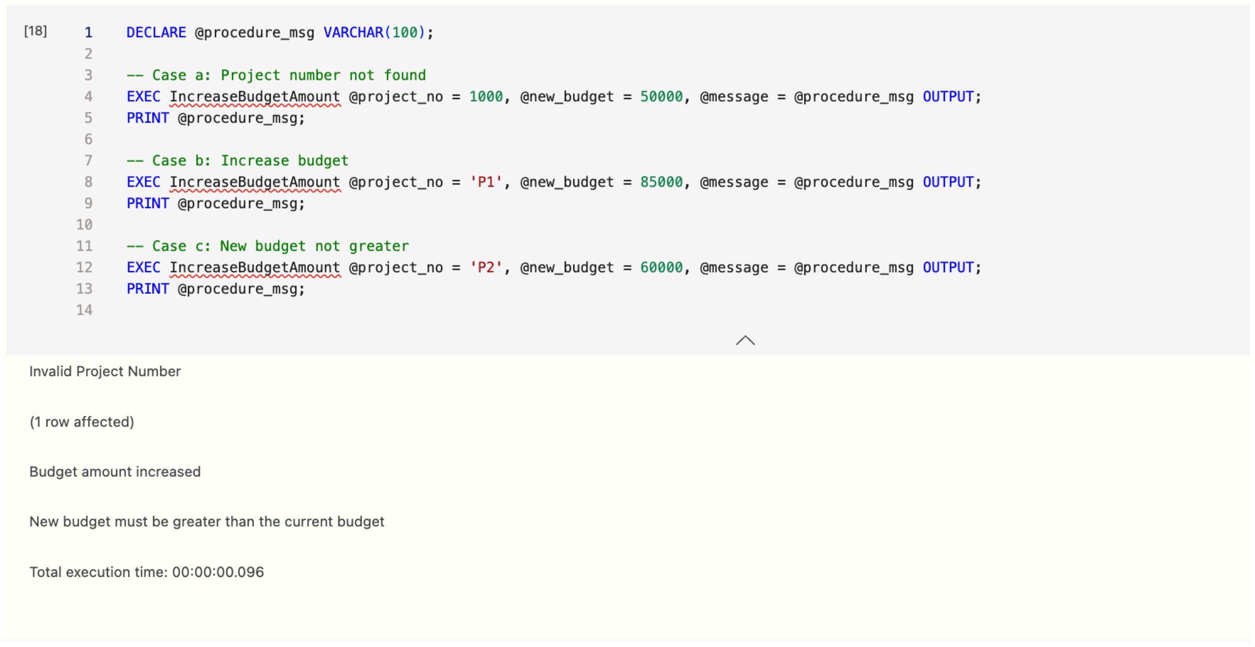
	project_no	project_name	budget
1	p1	Inventory system	75000
2	p2	CRM system	90000
3	p3	Order Management	135000
4	p4	Accounting System	202500
5	p5	ERP system	303750
6	p6	Data Warehouse	455625
7	p7	corporate website	683437.5

```
DECLARE @procedure_msg VARCHAR(100);

-- Case a: Project number not found
EXEC IncreaseBudgetAmount @project_no = 1000, @new_budget = 50000, @message =
@procedure_msg OUTPUT;
PRINT @procedure_msg;

-- Case b: Increase budget
EXEC IncreaseBudgetAmount @project_no = 'P1', @new_budget = 85000, @message =
@procedure_msg OUTPUT;
PRINT @procedure_msg;
```

```
-- Case c: New budget not greater
EXEC IncreaseBudgetAmount @project_no = 'P2', @new_budget = 60000, @message =
@procedure_msg OUTPUT;
PRINT @procedure_msg;
```



[18]

```
DECLARE @procedure_msg VARCHAR(100);

-- Case a: Project number not found
EXEC IncreaseBudgetAmount @project_no = 1000, @new_budget = 50000, @message = @procedure_msg OUTPUT;
PRINT @procedure_msg;

-- Case b: Increase budget
EXEC IncreaseBudgetAmount @project_no = 'P1', @new_budget = 85000, @message = @procedure_msg OUTPUT;
PRINT @procedure_msg;

-- Case c: New budget not greater
EXEC IncreaseBudgetAmount @project_no = 'P2', @new_budget = 60000, @message = @procedure_msg OUTPUT;
PRINT @procedure_msg;
```

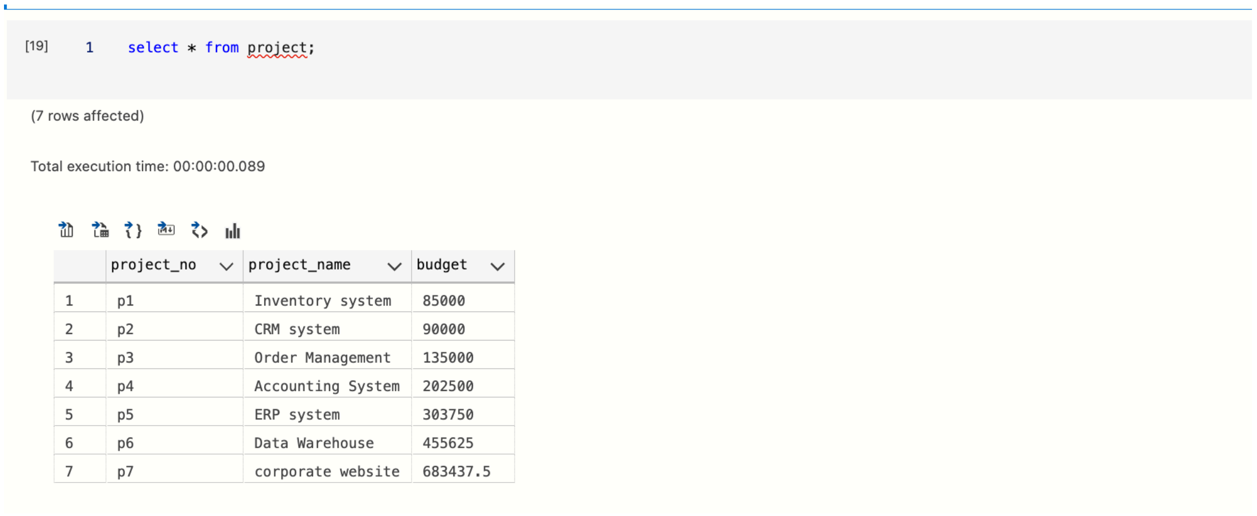
```
Invalid Project Number

(1 row affected)

Budget amount increased

New budget must be greater than the current budget

Total execution time: 00:00:00.096
```



[19]

```
select * from project;
```

```
(7 rows affected)

Total execution time: 00:00:00.089
```

	project_no	project_name	budget
1	p1	Inventory system	85000
2	p2	CRM system	90000
3	p3	Order Management	135000
4	p4	Accounting System	202500
5	p5	ERP system	303750
6	p6	Data Warehouse	455625
7	p7	corporate website	683437.5

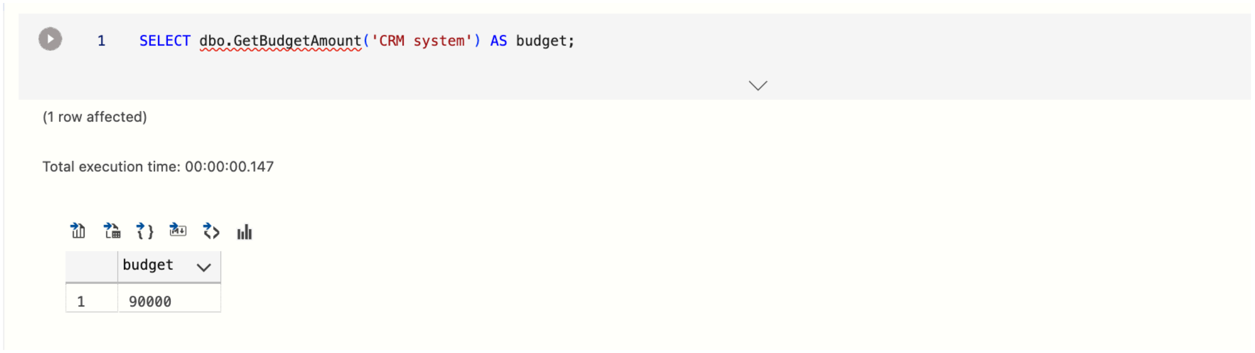
3) **Create a User Defined Function GetBudgetAmount which takes @project_name and returns the budget for a given project. If it cannot find the record it returns NULL. Show a SQL example of a function being used to the budget for "CRM system"**

```
CREATE FUNCTION GetBudgetAmount
(
   @project_name VARCHAR(100)
)
RETURNS FLOAT
AS
BEGIN
   DECLARE @budget FLOAT;

   SELECT @budget = budget
   FROM project
   WHERE project_name = @project_name;

   RETURN @budget;
END;


SELECT dbo.GetBudgetAmount('CRM system') AS budget;
```



```
SELECT dbo.GetBudgetAmount('CRM system') AS budget;
```

```
(1 row affected)

Total execution time: 00:00:00.147
```

	budget
1	90000

4) **What is the name of a single logical operation on the data to satisfy ACID property?**

It is a transaction. A transaction is a set of operations that is performed on a database as a single unit. i.e. either all the statements in the transaction will be executed or none will be executed. This atomicity ensures the consistency in data and maintain reliability of database.

5) **Which ACID property does the following DDLs satisfy?**
    a. **CREATE TABLE Customer (CustomerID int PRIMARY KEY, CustomerName varchar(100) NOT NULL)**

    It is a DDL where we are creating a new table customer with 2 attributes:
    1. CustomerID - primary key
    2. CustomerName

The ACID property it will follow is Atomicity. To create a table schema, the first thing to ensure is table create query is considered as single atomic statement. i.e. either the table is created with all of its attributes or not created at all.

6) **Which ACID property ensures the integrity of data reads?**

Isolation ensures the data integrity, a key ACID property in database transactions, stops multiple processes from messing with each other’s data. It makes sure transactions act on their own, with their uncommitted changes remain hidden from others until committed. This is often achieved through locking, which prevents any transaction from interfering with another's read or write operation on a specific piece of data.
In short, Isolation safeguards against "dirty reads" by guaranteeing a transaction only reads data that's been already committed by other transactions.

7) **Failure to write data to non-volatile memory violates which property?**
    a. **Atomicity**
    b. **Consistency**
    c. **Isolation**
    d. **Durability**

It violates Durability. The durability property guarantees that committed transactions are durable and persist even if the system crashes or loses power, thereby ensuring the reliability and fault tolerance of the database system.

It ensures that once a transaction is committed, and the changes made to the data are persisted permanently in non-volatile memory. If data changes made by a transaction are not successfully written to non-volatile memory and are subsequently lost due to a system failure, it violates the durability property.

8) **State the reasons why concurrency control needed?**

To maintain data consistency, concurrency control needed. concurrency control is a crucial mechanism in database management systems that ensures data integrity when multiple users or processes try to access and modify data concurrently.

There are multiple scenario where concurrency control maintains data integrity:

1. **To prevent dirty read:** A dirty read occurs when one transaction reads data that has been modified by another transaction but not yet committed. If the second transaction is later aborted, the first transaction has read and possibly acted upon invalid data.
2. **To prevent dirty write**: A dirty write happens when one transaction overwrites data that has been modified by another transaction but not yet committed. This can lead to inconsistency if the second transaction is later rolled back.
3. **Lost Update**: A lost update occurs when two transactions attempt to modify the same data concurrently, but one transaction's changes are overwritten by the other transaction before it commits. This results in one transaction's updates being lost and not reflected in the final database state.
4. **Deadlock Prevention**: Concurrency control algorithms help prevent deadlock situations where transactions are waiting indefinitely for resources held by other transactions. By detecting and resolving deadlocks, concurrency control ensures system availability and responsiveness.

9) What is the difference between a local transaction and a distributed transaction?

Feature	**Local Transaction**	**Distributed Transaction**
Database Scope	Involves a single database system or resource.	Involves multiple database systems or resources that are geographically distributed.
Geographic Scope	Local server	Can span across different server and it could be in different physical location.
Complexity	Less complex, managed by single database management system.	More Complex. It requires coordination among different database system to maintain ACID properties of a transaction.
Transactions	All operations within the transaction are performed on a single database	Operations within the transaction may span across multiple databases which can be located on different servers.

10) When should you use the SAVE TRANSACTION statement?

The save transaction statement isn't typically used for starting new transactions in modern database systems, but, commonly used to create specific points within an ongoing transaction called savepoints. These savepoints serve as markers that allow for rolling back to particular stages of the transaction in case of errors or issues. This feature is particularly useful in complex transactions where different stages need to be isolated for debugging or error handling purposes. Its use allows for more precise control over transaction behavior and error handling, but it may not be essential for straightforward transactional operations. There are many scenarios where save transaction can be useful:

For example:
Rolling Back to Specific Points - In complex transactions with multiple operations, SAVE TRANSACTION allows you to create named markers at specific points. If an error occurs later in the transaction, you can use ROLLBACK TRANSACTION to a specific savepoint, effectively undoing only the changes made since that point. This can be helpful for isolating the problematic section and preserving the work done before the error.

11) Discuss the difference between row-level and page-level locking.

Both are used for concurrency control and maintain data consistency.

Feature	**Row-level**	**Page-level**
Granularity	Locks a specific row or set of rows in a table. It offers greater concurrency since other transactions can access unlocked rows in the same table.	Locks an entire page of data in the database. A page typically contains multiple rows. This means that when a transaction accesses or modifies any data within a page, the entire page is locked,
Concurrency	It allows for higher concurrency as it locks only the specific rows being accessed. This enables other transactions to access different rows in the same table concurrently.	It allows low level concurrency. If one transaction acquires a lock on a page containing multiple rows, other transactions cannot access any of those rows until the lock is released.
Lock Overhead	Because each row individually requires its own lock, might incur slightly higher overhead due to the need to track and manage locks for individual rows.	As locks are acquired at the page level rather than the row level, it has lower overhead as managing locks for entire pages is less resource-intensive compared to individual rows.

12) Can a user explicitly influence the locking behavior of the system?

Although, user influence on locking behavior in database systems is generally limited, users can explicitly influence the locking behavior of a database system through various mechanisms provided by the database management system (DBMS).

Lock Hints: These are valuable tools that allow users to suggest specific locking strategies (shared locks, exclusive locks) for their queries or transactions. While the database system might not always follow these hints strictly, they can communicate user preferences and potentially influence locking behavior.

Transaction Isolation Levels: This is another powerful mechanism. By setting the isolation level (READ COMMITTED, REPEATABLE READ, SERIALIZABLE), users indirectly influence locking granularity and concurrency control. Higher isolation levels typically require stricter locks, impacting concurrency.

Locking Statements: Some advanced database systems offer explicit locking statements for manual control. Users can acquire and release locks on tables or rows using these statements. However, this approach can be complex and error-prone, so it's often used sparingly by experienced developers.

Optimistic Locking: This is an alternative approach that avoids traditional pessimistic locking (acquiring locks before data access). Optimistic locking acquires locks only during the commit phase. This allows for higher concurrency but requires conflict resolution mechanisms in case of conflicting updates.

Database Configuration Settings: While less direct, users can sometimes influence locking behavior through configuration settings related to locking, concurrency control, and transaction management. These settings can affect the default locking strategies employed by the database system.

In the end, we can say that, while the database system typically manages locking behavior automatically, users often could override or influence this behavior through explicit directives, transaction settings, and configuration options provided by the DBMS.