

**NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY**  
(AN AUTONOMOUS INSTITUTION, AFFILIATED TO VISVESVARAYA  
TECHNOLOGICAL UNIVERSITY  
BELGAUM, APPROVED BY AICTE & GOVT.OF KARNATAKA)



**CASE STUDY**

on

**DYNAMIC SQL**

*Submitted in partial fulfilment of the requirement for the award of Degree of  
Bachelor of Engineering  
in*

*Information Science and Engineering*

*Submitted by:*

Samprita.P	1NT20IS143
Sanket.D	1NT20IS146
Sneha.S	1NT20IS166

Under the Guidance of  
Dr.Tejaswini R Murgod, Associate Professor, Dept. of ISE, NMIT



Department of Information Science and Engineering

**(Accredited by NBA Tier-1)**

**NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY**  
(AN AUTONOMOUS INSTITUTION, AFFILIATED TO VISVESVARAYA  
TECHNOLOGICAL UNIVERSITY, BELGAUM)

Department of Information Science and Engineering

(Accredited by NBA Tier-1)



**CERTIFICATE**

This is to certify that the Project Report on “*Dynamic SQL*” is an authentic work carried out by **Samprita.P(1NT20IS143)**, **Sanket.D(1NT20IS146)**, **Sneha.S(1NT20IS166)**. Bonafide students of Nitte Meenakshi Institute of Technology, Bangalore in partial fulfilment for the award of the degree of Bachelor of Engineering in Information Science and Engineering of Visvesvaraya Technological University, Belagavi during the academic year 2021-2022. It is certified that all corrections and suggestions indicated during the internal assessment has been incorporated in the report.

**Internal Guide**

---

Dr. Tejaswini R Murgod  
Associate Professor  
Dept. ISE

**Signature of the HOD**

---

Dr. Mohan SG  
Professor, Head, Dept. ISE,  
NMIT Bangalore

## Acknowledgement

The satisfaction and euphoria that accompany the successful completion of any task would be incomplete without the mention of the people who made it possible, whose constant guidance and encouragement crowned our effort with success. I express my sincere gratitude to our Principal **Dr. H. C. Nagaraj**, Nitte Meenakshi Institute of Technology for providing facilities.

We wish to thank our HoD, **Dr. Mohan S. G.** for the excellent environment created to further educational growth in our college. We also thank him for the invaluable guidance provided which has helped in the creation of a better project.

I hereby like to thank our *Dr. Tejaswini R Murgod, Associate Professor* Department of Information Science & Engineering on their periodic inspection, time to time evaluation of the project and help to bring the project to the present form.

Thanks to our Departmental Project coordinators. We also thank all our friends, teaching and non-teaching staff at NMIT, Bangalore, for all the direct and indirect help provided in the completion of the project.

## TABLE OF CONTENTS

<b>Sl.no</b>	<b>Chapter Title</b>	<b>Page Number</b>
1	Abstract	5-6
2	Introduction	7
3	Implementation	8
4	When to use Dynamic SQL	9
5	Optimise Execution Dynamically	10
6	Case Study	11-12
7	Conclusion	13

## References

## Chapter-1

### ABSTRACT

Dynamic SQL enables you to write programs that reference SQL statements whose full text is not known until runtime. Before discussing dynamic SQL in detail, a clear definition of static SQL may provide a good starting point for understanding dynamic SQL. Static SQL statements do not change from execution to execution. The full text of static SQL statements are known at compilation, which provides the following benefits:

- Successful compilation verifies that the SQL statements reference valid database objects.
- Successful compilation verifies that the necessary privileges are in place to access the database objects.
- Performance of static SQL is generally better than dynamic SQL.

Because of these advantages, you should use dynamic SQL only if you cannot use static SQL to accomplish your goals, or if using static SQL is cumbersome compared to dynamic SQL. However, static SQL has limitations that can be overcome with dynamic SQL. You may not always know the full text of the SQL statements that must be executed in a PL/SQL procedure. Your program may accept user input that defines the SQL statements to execute, or your program may need to complete some processing work to determine the correct course of action. In such cases, you should use dynamic SQL. For example, consider a reporting application that performs standard queries on tables in a data warehouse environment where the exact table name is unknown until runtime. To accommodate the large amount of data in the data warehouse efficiently, you create a new table every quarter to store the invoice information for the quarter. These tables all have exactly the same definition and are named according to the starting month and year of the quarter. 3 for example INV\_01\_1997, INV\_04\_1997, INV\_07\_1997, INV\_10\_1997, INV\_01\_1998, etc. In such a case, you can use dynamic SQL in your reporting application to specify the table name at runtime. With static SQL, all of the data definition information, such as table definitions, referenced by the SQL statements in your program must be known at compilation. If the data definition changes, you must change and recompile the program. Dynamic SQL programs can handle changes in data definition information, because the SQL statements can change "on the fly" at runtime. Therefore, dynamic SQL is much more flexible than static SQL. Dynamic SQL enables you to write application code that is reusable

because the code defines a process that is independent of the specific SQL statements used. In addition, dynamic SQL lets you execute SQL statements that are not supported in static SQL programs, such as data definition language (DDL) statements. Support for these statements allows you to accomplish more with your PL/SQL programs.

## Chapter-2

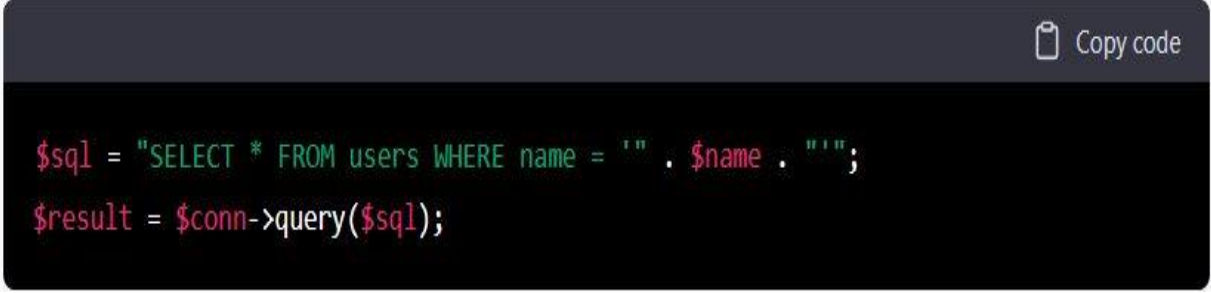
### INTRODUCTION

Dynamic SQL refers to the ability to generate and execute SQL statements at runtime, rather than hardcoding them into an application.

This can be useful in a variety of situations, such as when the structure or content of the SQL statements needs to be determined at runtime, when building a flexible query builder, or when dealing with a database schema that is subject to change.

There are several ways to implement dynamic SQL in different programming languages. One approach is to use string concatenation to construct the SQL statement as a string, and then execute the string using a database API.

For example:

A code block with a dark background and light-colored text. In the top right corner, there is a clipboard icon followed by the text "Copy code". The code itself is written in a syntax-highlighted style with red, green, and black text.

```
$sql = "SELECT * FROM users WHERE name = '" . $name . "'";  
$result = $conn->query($sql);
```

## Chapter-3

### IMPLEMENTATION

There are several ways to implement dynamic SQL in different programming languages.

One approach is to use string concatenation to construct the SQL statement as a string, and then execute the string using a database API.

While the approach is simple, it can be prone to SQL injection vulnerabilities if user input is not properly sanitized.

To address this issue, another approach is to use placeholders in the SQL statement and then bind values to the placeholders at runtime. This can help prevent SQL injection attacks by ensuring that user input is properly escaped.

For example:

A code block with a dark background and light-colored text. In the top right corner, there is a clipboard icon and the text "Copy code". The code is a PHP snippet demonstrating a prepared statement using PDO. It starts with a SQL query string containing a placeholder, then prepares the statement, binds a parameter, and finally executes it.

```
$sql = "SELECT * FROM users WHERE name = ?";  
$stmt = $conn->prepare($sql);  
$stmt->bind_param('s', $name);  
$result = $stmt->execute();
```

In addition to the benefits of flexibility and adaptability, dynamic SQL can also be useful for improving the performance of an application. For instance, it can allow you to build more efficient queries by only selecting the columns that are actually needed, rather than selecting all columns as is often done in static SQL.



## **Chapter-4**

### **WHEN TO USE DYNAMIC SQL**

One should use dynamic SQL in cases where static SQL does not support the operation you want to perform.

In cases where you do not know the exact SQL statements that must be executed by a PL/SQL procedure.

These SQL statements may depend on user input, or they may depend on processing work done by the program.

## Chapter-5

### OPTIMIZE EXECUTION DYNAMICALLY

If we use static SQL, we must decide at compilation how you want to construct your SQL statements, whether to have hints in your statements, and, if you include hints, exactly which hints to have. However, you can use dynamic SQL to build a SQL statement in a way that optimizes the execution and/or concatenates the hints into a SQL statement dynamically. This allows you to change the hints based on your current database statistics, without requiring recompilation.

**For example, the following procedure uses a variable called `a_hint` to allow users to pass a hint option to the `SELECT` statement:**

```
CREATE OR REPLACE PROCEDURE  
query_emp (a_hint VARCHAR2) AS  
TYPE cur_typ IS REF CURSOR;  
c cur_typ;  
BEGIN  
OPEN c FOR 'SELECT ' || a_hint ||  
' empno, ename, sal, job FROM emp WHERE empno = 7566'; -- process  
END; /
```

## Chapter-6

### CASE STUDY

#### Problem:

ACME Corporation is a software development company that specializes in building custom enterprise applications for its clients. One of the applications they developed was a *CRM (customer relationship management)* system for a large retail chain. The retail chain had a complex database schema with many tables, and they needed the CRM system to be able to generate flexible queries to extract and analyze data from the database.

#### Solution:

To meet these requirements, the ACME team decided to use dynamic SQL in their CRM system. They wrote a set of functions that allowed users to build custom SQL queries using a simple API, without having to write any raw SQL themselves.

```
CREATE TABLE dbo.Customer
(
    customerid INT IDENTITY PRIMARY KEY,
    firstname VARCHAR(40) NOT NULL,
    lastname VARCHAR(40) NOT NULL,
    statecode VARCHAR(2) NOT NULL,
    totalsales money NOT NULL DEFAULT 0.00
)
```

```
INSERT INTO dbo.Customer (firstname, lastname, statecode, totalsales)
SELECT 'Thomas', 'Jefferson', 'VA', 100.00
```

```
INSERT INTO dbo.Customer (firstname, lastname, statecode, totalsales)
SELECT 'John', 'Adams', 'MA', 200.00
```

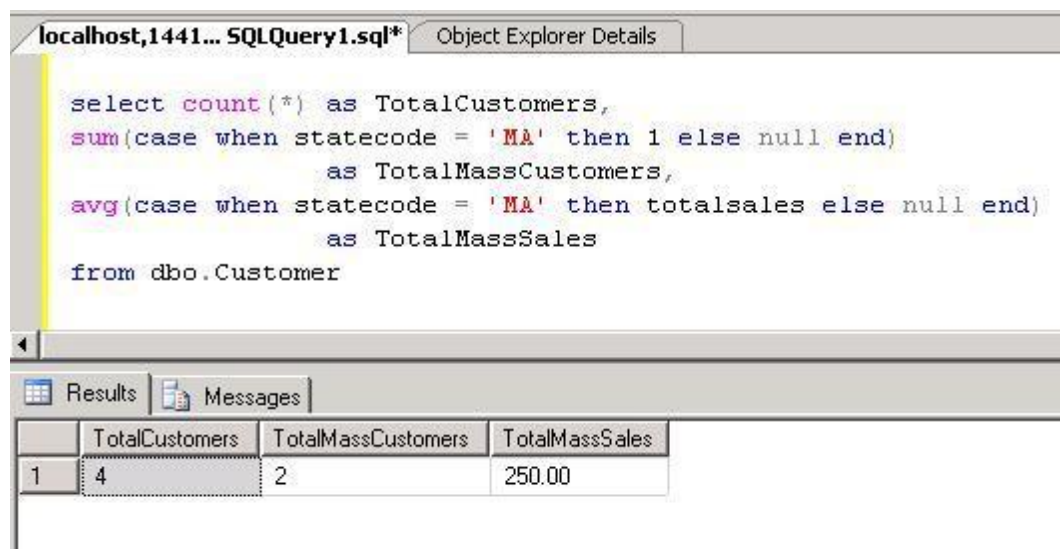
```
INSERT INTO dbo.Customer (firstname, lastname, statecode, totalsales)
SELECT 'Paul', 'Revere', 'MA', 300.00
```

```
INSERT INTO dbo.Customer (firstname, lastname, statecode, totalsales)
SELECT 'Ben', 'Franklin', 'PA', 400.00
GO
```

A requirement has come in where we need to report on the total number of all customers, the total number of all 'M' customers, and an average of all sales made by all 'M' customers. We could limit the query to just 'M' customers but that would make it cumbersome to get our count of total customers. To solve this problem, you can write the query to use an expression within the aggregate functions to get 'M' specific information:

```
SELECT COUNT(*) AS TotalCustomers,  
SUM(WHEN statecode = 'M' THEN 1 ELSE NULL END) AS  
TotalMCustomers,  
AVG( WHEN statecode = 'M' THEN totalsales ELSE NULL END) AS  
TotalMSales  
FROM dbo.Customer
```

Since NULL values are discarded when performing aggregate functions, we can easily get the required totals.



The screenshot shows a SQL Server Enterprise Manager window with a query editor and a results grid. The query editor contains the following SQL code:

```
select count(*) as TotalCustomers,  
sum(case when statecode = 'MA' then 1 else null end)  
as TotalMassCustomers,  
avg(case when statecode = 'MA' then totalsales else null end)  
as TotalMassSales  
from dbo.Customer
```

The results grid shows the following data:

	TotalCustomers	TotalMassCustomers	TotalMassSales
1	4	2	250.00

## **Chapter-7**

### **CONCLUSION**

We understand that it is important to use dynamic SQL carefully, as it can also introduce security risks and complexity to an application.

In particular, it is important to properly sanitize user input to prevent SQL injection attacks, and to thoroughly test the dynamic SQL code to ensure that it is correct and secure.

Overall, dynamic SQL can be a powerful tool for building flexible and adaptable database-driven applications, but it is important to use it judiciously and to consider the potential trade-offs and risks involved.

## **References**

<https://www.oracle.net>

<https://www.sqlservertutorial.net>