UNIVERSITY OF MINES AND TECHNOLOGY

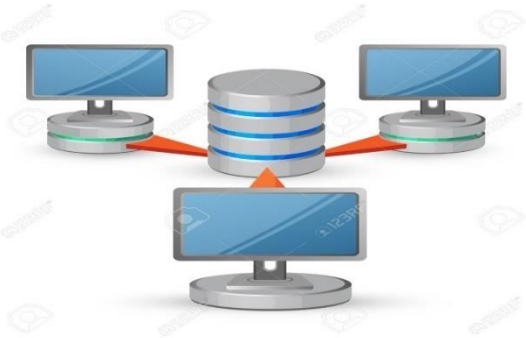
TARKWA

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LECTURE NOTES

CE 280 Advanced Database Management Systems



COMPILED BY

Dr William Akotam Agangiba

PhD (UMaT, Ghana), MSc(TSTU, Russia), BSc(TSTU, Russia), IEEE Member

EDITED BY

Dr Mrs Millicent Agangiba

PhD (South Africa), (MSc, Russia) SMIEEE, PE-IET(GH), MACM. MAIS. IAENG. MISOC. MWISWB

JUNE, 2022

TABLE OF CONTENT

Course Objective

This course is intended to provide an understanding of the current theory and practice of database management systems. The course provides a solid technical overview of database management systems, using a current database product as a case study. In addition to technical concerns, more general issues are emphasized. These include data independence, integrity, security, recovery, performance, database design principles, and database administration.

Course Content

Open database connectivity (ODBC), Triggers, procedural SQL (PL/SQL) and stored procedures, XML databases, distributed databases, Web-based database application systems, relational algebra and joins, query processing and optimization, Data mining and data warehouse.

Reference Materials:

a) Jason, G. W. (2010), *Beginning PhP and MySQL: From Novice to Professional*, Associated Press, 4th edition, 835 pp., ISBN: 978-1-4302-3114-1.
b) Yank, K. (2012), *PhP & MySQL: Novice to Ninja*, SitePoint Pty. Ltd., 5th edition, 524 pp., ISBN: 978-0-7356-6609-2.
c) Ellis, G. (2014), *Getting Started with SQL Server 2014 Administration*, Packt Publishing Ltd, 1st edition, 106 pp., ISBN: 978-1-78217-241-3.
d) Sarka, D., Lah, M. and Jerkic, G. (2012), *Implementing a Data Warehouse with Microsoft SQL Server 2012*, O'Reilly Media Inc., 1st edition, 848 pp., ISBN: 978-0-7356-6609-2.

Course Presentation

The course is presented through lectures and Laboratory Hands-on Practice supported with a hand-out. During the Lab sessions, students are guided to solve practical problems with varying degrees of difficulty. Students are also given practical exercise to solve on their own and submit solutions in the form of assignments. The student can best understand and appreciate the subject by attending all lectures and

laboratory work, by practising, reading references and hand-outs and by completing all assignments on schedule.

Course Assessment

	Factor	Weight	Location	Date	Time
Grading System	Quizzes (3)	15 %	In class	To Be Announced (TBA)	1 Hour/Each
	Assignments	15%	SD		
	Attendance	10 %	In class	Random	
	Final Exam	60 %	(TBA)	To Be Announced (TBA)	Practical (2hrs)
					Written (2hrs)

80-100% **A**
70-79.9% **B**
60-69.9% **C**
50-59.9% **D**
0-49.9% **FAIL**

Attendance

According to UMaT rules and regulations, attendance is mandatory for every student. A random attendance shall be taken to constitute 10% of the total semester mark. The only acceptable excuse for absence is one authorized by the Dean of Students on their prescribed form. However, a student can also ask permission from me to be absent from a particular class with a tangible reason. A student who misses more than 50% of the total attendance marked WOULD not be allowed to take the final exams.

1. DATABASE PROGRAMMABILITY

1.1 Stored Procedures

A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again. So, if you have an SQL query that you write over and over again, save it as a stored procedure, and then just call it to execute it. You can also pass parameters to a stored procedure, so that the stored procedure can act based on the parameter value(s) that is passed.

Syntax:

```
CREATE PROCEDURE procedure_name
AS
sql_statement
GO;
```

To Execute a stored procedure, simple using the following syntax in the command Window:

```
EXEC procedure_name;
```

Consider Table 1:

Table 1. Pilots of BabyJet Airlines

PID	Name	Age	Nationality	Destination
1	John	45	Ghanaian	Dubai
2	Quainoo	50	Ghanaian	KIA
3	Haile	35	Ethiopian	Cape Town
4	Gregoriev	40	Russian	Kieve
5	Tyler	45	American	Abijan

A query to fetch names of pilots, their nationalities and flight destinations can be written as follows:

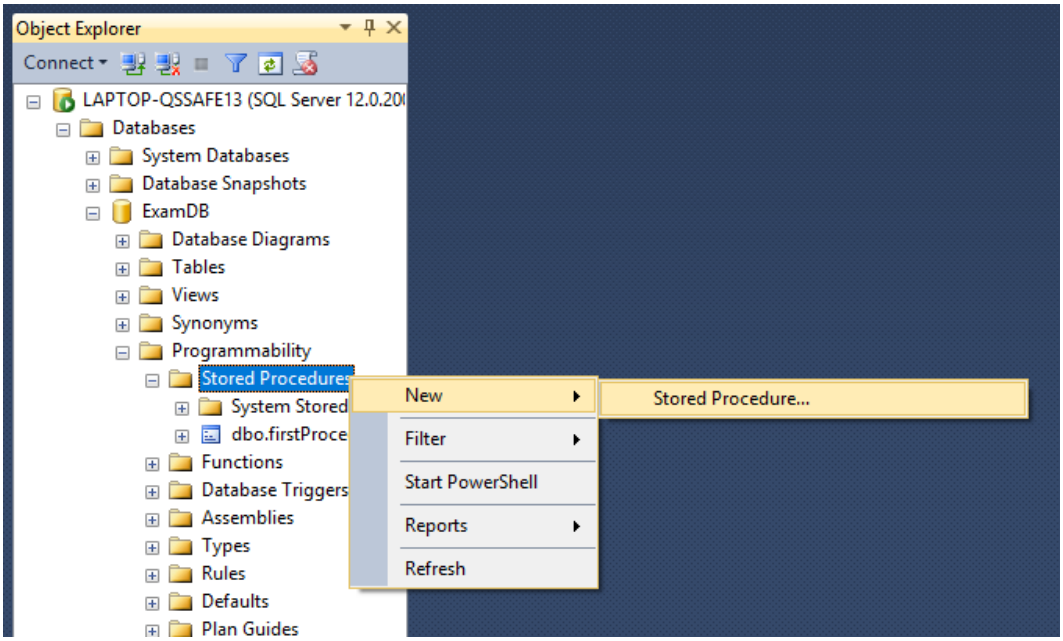
```
SELECT Name, Nationality,Destination
FROM
ExamDB.dbo.Pilot
ORDER BY
Name;
```

If this were a regular routine, it is worth organising such a query into a stored procedure as follows:

```
CREATE PROCEDURE firstProcedure
AS
BEGIN
SELECT
 Name,
 Nationality,
 Destination
FROM
 ExamDB.dbo.Pilot
ORDER BY
 Name;
END
GO
```

Writing Stored procedures in MS SQL server

i. On the Object Explorer, click on the plus (+) sign beside the database for which you want to create the stored procedure.
ii. Click on the plus (+) sign beside "programmability" to Expand it.
iii. Right-click on stored "Stored Procedures" and select "New Stored Procedure". See the figure below



i. This gives access to the editor for you to write the Stored Procedure.

ii. Once the stored procedure is written and Saved, it becomes part of the sql server and appears under programmability of the given databases.

iii. To execute it, open a new query window and type: EXEC *procedure_name*;

To delete a stored procedure, open a query window and type DROP PROCEDURE *procedure_name*;

1.1.1 Procedures with Parameters

Sometimes, it is useful to create stored procedures to allow users to specify values or make choices when calling such procedures. Such procedures are commonly termed as stored procedures with parameters. They can take one or more parameters depending on the choice of the programmer. The following stored procedure takes only one parameter. Anytime the procedure runs, it outputs only Names, Nationalities of Pilots whose ages are less than whatever value will be specified when the procedure is called.

```
CREATE PROCEDURE firstParamProcedure(@choiceAge as Integer)
AS
BEGIN
SELECT
 Name,
 Nationality,
 Age
FROM
 ExamDB.dbo.Pilot
WHERE
 Age <= @choiceAge
ORDER BY
 Name;
END
GO
```

An example of call of the procedure is shown next:

```
Exec firstParamProcedure 40;
```

The output of this call is:

	Name	Nationality	Age
1	Gregoriev	Russian	40
2	Haile	Ethiopian	35

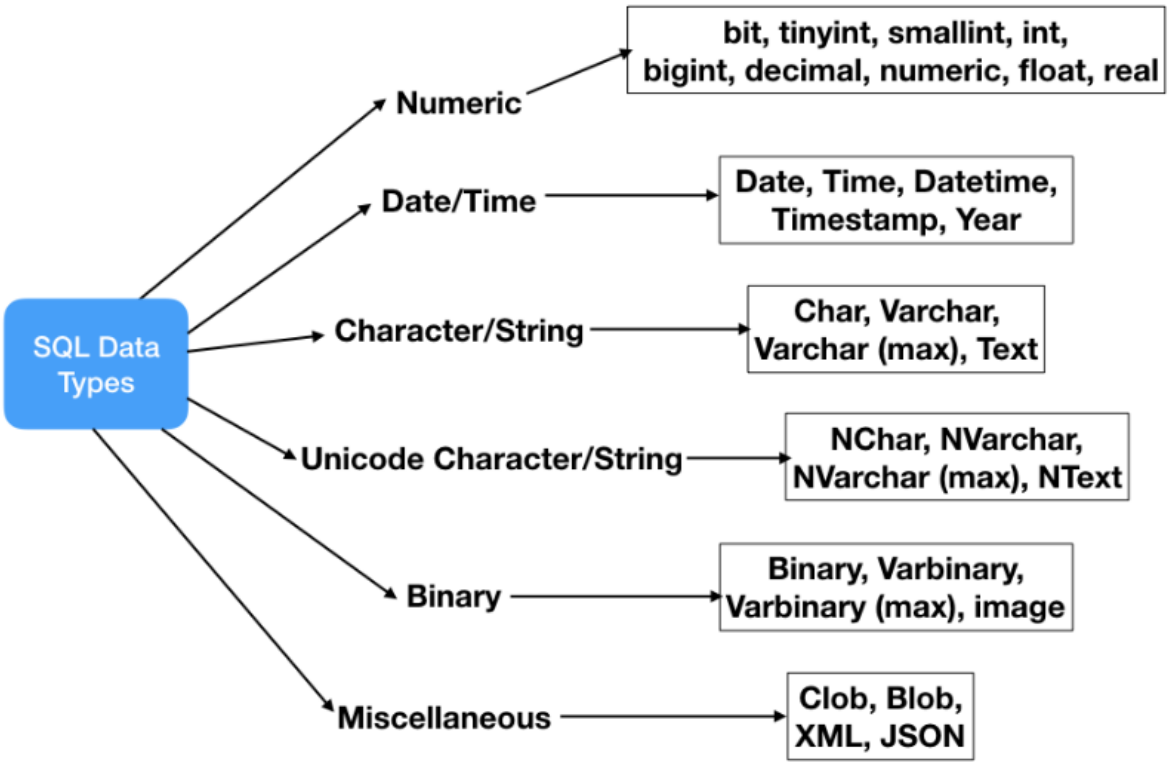
Every parameter must start with the @ sign. The AS Integer keywords specify the data type of the @choiceAge parameter. The parameter must be surrounded by the opening and closing brackets.

Second, we used @choiceAge parameter in the WHERE clause of the SELECT statement to filter only the pilots whose ages are lesser than or equal to the @@choiceAge.

Stored procedures can take one or more parameters. The parameters are separated by commas.

1.1.2 Variables

Variables are used in SQL statements for the purpose of holding data in various situations for various purposes. To use a variable, it is necessary to first declare it with the right datatype.



SQL Variables are declared with the following syntax:

```
DECLARE @variable_Name DATATYPE
```

You can replace variable_Name with any standard variable name of your choice. An example of a variable declaration is given below:

```
DECLARE @myAge SMALLINT;
```

The variable name must start with the @ sign. In this example, the data type of the @myAge variable is SMALLINT. By default, when a variable is declared, its value is set to NULL. Between the variable name and data type, you can use the optional AS keyword as follows:

```
DECLARE @myAge As SMALLINT;
```

To declare multiple variables, you separate variables by commas:

Assigning a value to a variable

The SET keyword is used when assigning a value to statement. For example, the following statement assigns 28 to the @myAge variable:

```
SET @myAge =28;
```

The following query makes use of variables as discussed above:

```
DECLARE @myAge AS SMALLINT;
SET @myAge = 35;
SELECT Name, Age, Nationality,Destination
FROM
ExamDB.dbo.Pilot
WHERE
Age = @myAge

ORDER BY
Name;
```

Storing query result in a variable

The following query counts the number of pilots who are Ghanaians and stores the results in a variable and outputs it.

```
DECLARE @Total INT;
SET @Total =(SELECT
 COUNT(*)
```

```
FROM
 ExamDB.dbo.Pilot
 WHERE
 Nationality='Ghanaian'
);
PRINT @Total;
```

The 'PRINT' clause could be re-written to make more meaning as follows:

```
PRINT 'The number of products is ' + CAST(@Total AS
VARCHAR(MAX));
```

Selecting a record into variables

```
DECLARE
   @pilotName As VARCHAR(25),
   @pilotNationality As VARCHAR(25);

SELECT
   @pilotName = Name,
   @pilotNationality = Nationality
FROM
   ExamDB.dbo.Pilot
WHERE
   Nationality = 'Ghanaian';
PRINT  @pilotName
PRINT @pilotNationality
```

1.2 Database Triggers

A database trigger is procedural code that is automatically executed in response to certain events on a particular table or view in a database. The trigger is mostly used for keeping the integrity of the information on the database. For example, when a new record (representing a new worker) is added to the employees table, new records should be created also in the tables of the taxes, vacations and salaries.

The need for trigger and the usage

Triggers are commonly used to:

- prevent changes (e.g., prevent an invoice from being changed after it's been mailed out)
- log changes (e.g., keep a copy of the old data)
- audit changes (e.g., keep a log of the users and roles involved in changes)
- enhance changes (e.g., ensure that every change to a record is time-stamped by the server's clock, not the client's)
- enforce business rules (e.g., require that every invoice have at least one line item)
- execute business rules (e.g., notify a manager every time an employee's bank account number changes)
- replicate data (e.g., store a record of every change, to be shipped to another database later)
- enhance performance (e.g., update the account balance after every detail transaction, for faster queries)

The examples above are called Data Manipulation Language (DML) triggers because the triggers are defined as part of the Data Manipulation Language and are executed at the time the data are manipulated. Some systems also support non-data triggers, which fire in response to Data Definition Language (DDL) events such as creating tables, or runtime or and events such as logon, commit, and rollback. Such DDL triggers can be used for auditing purposes.

The following are major features of database triggers and their effects:

- triggers do not accept parameters or arguments (but may store affected-data in temporary tables)

- triggers cannot perform commit or rollback operations because they are part of the triggering SQL statement (only through autonomous transactions)
- triggers can cancel a requested operation
- triggers can cause mutating table errors

1.2.1 Creating a Database Trigger

CREATE TRIGGER command is used to create a database trigger. The following details are to be given at the time of creating a trigger.

_ Name of the trigger

_ Table to be associated with

_ When trigger is to be fired - before or after

_ Command that invokes the trigger - UPDATE, DELETE, or INSERT

_ Whether row-level trigger or not

_ Condition to filter rows.

_ PL/SQL block that is to be executed when trigger is fired.

The following is the syntax of CREATE TRIGGER command.

```
CREATE [OR REPLACE] TRIGGER trigername
{BEFORE | AFTER}
{DELETE | INSERT | UPDATE [OF columns]}
[OR {DELETE | INSERT |UPDATE [OF columns]}]...
ON table
[FOR EACH ROW [WHEN condition]]
[REFERENCING [OLD AS old] [NEW AS new]]
PL/SQL block
```

SQL Triggers

The SQL CREATE TRIGGER statement provides a way for the database management system to actively control, monitor, and manage a group of tables whenever an insert, update, or delete operation is performed. The statements specified

in the SQL trigger are executed each time an SQL insert, update, or delete operation is performed. An SQL trigger may call stored procedures or user-defined functions to perform additional processing when the trigger is executed.

Unlike stored procedures, an SQL trigger cannot be directly called from an application. Instead, an SQL trigger is invoked by the database management system on the execution of a triggering insert, update, or delete operation. The definition of the SQL trigger is stored in the database management system and is invoked by the database management system, when the SQL table, that the trigger is defined on, is modified.

An SQL trigger can be created by specifying the CREATE TRIGGER SQL statement. The statements in the routine-body of the SQL trigger are transformed by SQL into a program (*PGM) object.

2.2.2 Types of Triggers

1. DML Triggers
    - AFTER Triggers
    - INSTEAD OF Triggers
2. DDL Triggers
3. CLR Triggers

DML Triggers

These triggers are fired when a Data Manipulation Language (DML) event takes place. These are attached to a Table or View and are fired only when an INSERT, UPDATE and/or DELETE event occurs. The trigger and the statement that fires it are treated as a single transaction. Using this we can cascade changes in related tables, can do check operations for satisfying some rules and can get noticed through

firing Mails. We can even execute multiple triggering actions by creating multiple Triggers of same action type on a table. We have to specify the modification action(s) at the Table level that fires the trigger when it is created.

AFTER Triggers

As the name specifies, AFTER triggers are executed after the action of the INSERT, UPDATE, or DELETE statement is performed. AFTER triggers can be specified on tables only, here is a sample trigger creation statement on the Users table.

Create table Person (age int);

```
CREATE TRIGGER PersonCheckAge
AFTER INSERT OR UPDATE OF age ON Person
FOR EACH ROW
BEGIN
   IF (:new.age < 0) THEN
      RAISE_APPLICATION_ERROR (-20000, 'no negative age
allowed');
   END IF;
END;
```

If we attempted to execute the insertion:

Insert into Person values (-3);

We would get the error message:

ERROR at line 1:
ORA-20000: no negative age allowed
ORA-06512: at "MYNAME.PERSONCHECKAGE", line 3

ORA-04088: error during execution of trigger 'MYNAME.PERSONCHECKAGE'
and nothing would be inserted.

Instead of Triggers

INSTEAD OF triggers are executed in place of the usual triggering action. INSTEAD OF triggers can also be defined on views with one or more base tables, where they can extend the types of updates a view can support.

DDL Triggers

This type of triggers, like regular triggers, fire stored procedures in response to an event. They fire in response to a variety of Data Definition Language (DDL) events. These events are specified by the T-SQL statements that are start with the keywords CREATE, ALTER, and DROP. Certain stored procedures that perform DDL-like operations can also fire this. These are used for administrative tasks like auditing and regulating database operations.

DDL triggers can fire in response to a Transact-SQL event processed in the current database, or on the current server. The scope of the trigger depends on the event. For example, a DDL trigger created to fire in response to a CREATE_TABLE event can do so whenever a CREATE_TABLE event occurs in the database, or on the server instance. A DDL trigger created to fire in response to a CREATE_LOGIN event can do so only when a CREATE_LOGIN event occurs in the server.

In the following example, DDL trigger safety will fire whenever a DROP_TABLE or ALTER_TABLE event occurs in the database.

```
CREATE TRIGGER safety
ON DATABASE
FOR DROP_TABLE, ALTER_TABLE
AS
  PRINT 'You must disable Trigger "safety" to drop or alter tables!'
  ROLLBACK
;
```

CLR Triggers - A CLR triggers can be any of the above, e.g. can be a DDL or DML one or can also be an AFTER or INSTEAD OF trigger. Here we need to execute one

or more methods written in managed codes that are members of an assembly created in the .Net framework. Again, that assembly must be deployed in SQL Server 2005 using CREATE assembly statement

1.3 Functions

SQL Server scalar function takes one or more parameters and returns a single value. The scalar functions help simplify code. For example, you may have a complex calculation that appears in many queries. Instead of including the formula in every query, you can create a scalar function that encapsulates the formula and uses it in the queries.

1.3.1 Creating a scalar function

To create a scalar function, you use the CREATE FUNCTION statement as follows

```
CREATE FUNCTION [schema_name.]function_name (parameter_list)
RETURNS data_type AS
BEGIN
  statements
  RETURN value
END
```

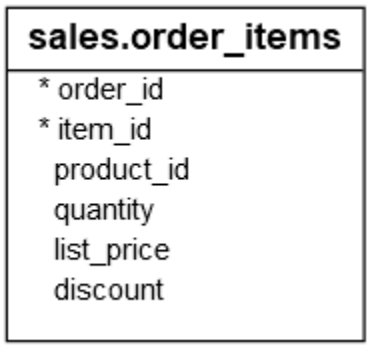
In this syntax:

- First, specify the name of the function after the CREATE FUNCTION keywords. The schema name is optional. If you don't explicitly specify it, SQL Server uses dbo by default.
- Second, specify a list of parameters surrounded by parentheses after the function name.
- Third, specify the data type of the return value in the RETURNS statement.
- Finally, include a RETURN statement to return a value inside the body of the function.

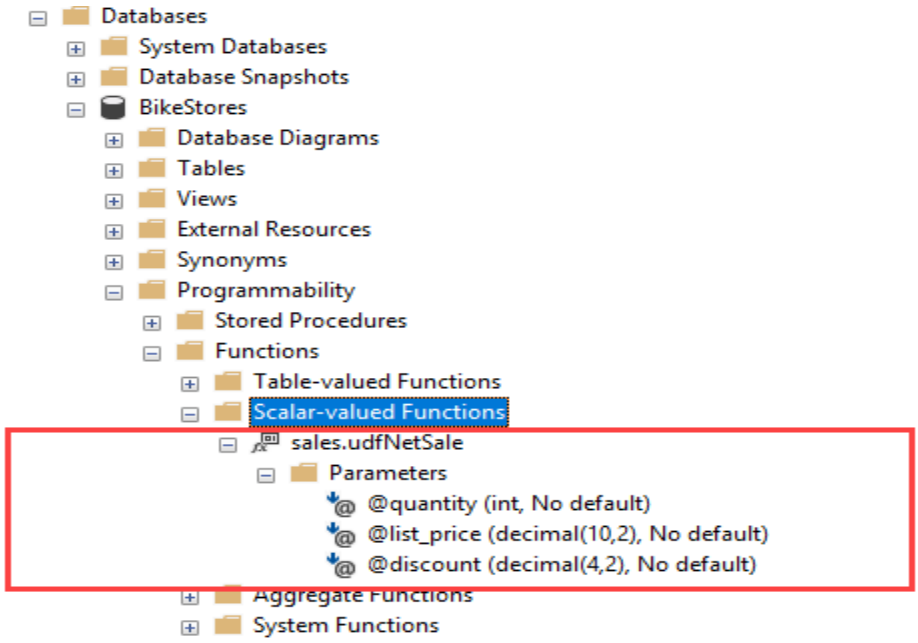
The following example creates a function that calculates the net sales based on the quantity, list price, and discount:

```
CREATE FUNCTION sales.udfNetSale(
    @quantity INT,
    @list_price DEC(10,2),
    @discount DEC(4,2)
)
RETURNS DEC(10,2)
AS
BEGIN
    RETURN @quantity * @list_price * (1 - @discount);
END;
```

Reference Database Table:



After creating the scalar function, you can find it under **Programmability > Functions > Scalar-valued Functions** as shown in the following picture:



1.3.2 Calling a scalar function

You call a scalar function like a built-in function. For example, the following statement demonstrates how to call the udfNetSale function:

The following example illustrates how to use the sales.udfNetSale function to get the net sales of the sales orders in the order_items table:

```
SELECT
   order_id,
   SUM(sales.udfNetSale(quantity, list_price, discount)) net_amount
FROM
   sales.order_items
GROUP BY
   order_id
ORDER BY
   net_amount DESC;
```

The following picture shows the partial output:

order_id	net_amount
1541	29147.02
937	27050.71
1506	25574.95
1482	25365.43
1364	24890.62
930	24607.02
1348	20648.95
1334	20509.42
973	20177.74
1118	19329.94
1349	18853.35
1115	18670.92
1277	18553.72

1.3.3 Modifying a scalar function

To modify a scalar function, you use the ALTER instead of the CREATE keyword. The rest statements remain the same:

```
ALTER FUNCTION [schema_name.]function_name (parameter_list)
  RETURN data_type AS
  BEGIN
    statements
    RETURN value
  END
```

Note that you can use the CREATE OR ALTER statement to create a user-defined function if it does not exist or to modify an existing scalar function:

```
CREATE OR ALTER FUNCTION [schema_name.]function_name (parameter_list)
    RETURN data_type AS
    BEGIN
      statements
      RETURN value
    END
```

To remove an existing scalar function, you use the DROP FUNCTION statement:

```
DROP FUNCTION [schema_name.]function_name;
```

For example, to remove the sales.udfNetSale function, you use the following statement:

```
DROP FUNCTION sales.udfNetSale;
```

CHAPTER QUESTIONS

1. What do you understand by a stored procedure?
2. Mention the uses of stored procedures.
3. What are the types of stored procedures in an SQL server?
4. What are the advantages of using a stored procedure?
5. What are the differences between stored procedure and view in SQL server?
6. When would you use the stored procedures or functions?
7. What is a Trigger?
8. What are the types of Triggers?
9. What is the difference between trigger and stored procedure?
10. What query will be used to get the list of triggers in a database?
11. What is a DML Trigger in SQL Server?
12. DML triggers can be again classified into 2 types, name them.
13. How do you Enable, Disable and Drop Triggers in SQL Server?
14. What is the difference between InsteadOf and After/For trigger?
15. You have a table with some of the temperatures in Celsius of some patients.

    Create a function or a stored procedure to get the convert Fahrenheit to Celsius.

2. DATABASES TRANSACTIONS

A transaction is the execution of a program that accesses or changes the content of a database. It is a logical unit of work (LUW) on the database that is either completed in its entirety (COMMIT) or not done at all. In the latter case, the transaction has to clean up its own mess, known as ROLLBACK. A transaction could be an entire program, a portion of a program or a single command.

The concept of a transaction is inherently about organising functions to manage data. A transaction may be distributed (available on different physical systems or organised into different logical subsystems) and/or use data concurrently with multiple users for different purposes. Online transaction processing (OLTP) systems support a large number of concurrent transactions without imposing excessive delays.
Transactions in a database environment have two main purposes:

- To provide reliable units of work that allow correct recovery from failures and keep a database consistent even in cases of system failure, when execution stops (completely or partially) and many operations upon a database remain uncompleted, with unclear status.
- To provide isolation between programs accessing a database concurrently. Without isolation the programs' outcomes are possibly erroneous.

For recovery purposes, a system always keeps track of when a transaction starts, terminates, and commits or aborts. Hence, the recovery manager keeps track of the following transaction states and operations:

• BEGIN_TRANSACTION: This marks the beginning of transaction execution.

• READ or WRITE: These specify read or write operations on the database items that are executed as part of a transaction.

• END_TRANSACTION: This specifies that read and write operations have ended and marks the end limit of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates concurrency control, or for some other reason (rollback).

• COMMIT_TRANSACTION: This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.

• ROLLBACK (or ABORT): This signals the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone

A database transaction, by definition, must be atomic, consistent, isolated and durable. Database practitioners often refer to these properties of database transactions using the acronym **ACID.**

2.1 Transaction ACID Rules

Every database transaction must obey the following rules:

Atomicity - states that database modifications must follow an "all or nothing" rule. Each transaction is said to be "atomic." If one part of the transaction fails, the entire transaction fails. It is critical that the database management system maintain the

atomic nature of transactions in spite of any DBMS, operating system or hardware failure.

Consistency - states that only valid data will be written to the database. If, for some reason, a transaction is executed that violates the database's consistency rules, the entire transaction will be rolled back and the database will be restored to a state consistent with those rules. On the other hand, if a transaction successfully executes, it will take the database from one state that is consistent with the rules to another state that is also consistent with the rules.

Isolation - Transactions cannot interfere with each other. Moreover, an incomplete transaction is not visible to another transaction. Providing isolation is the main goal of concurrency control.

Durability - ensures that any transaction committed to the database will not be lost. Durability is ensured through the use of database backups and transaction logs that facilitate the restoration of committed transactions in spite of any subsequent software or hardware failures.

2.2 Transactions Example & ACID Properties

Consider a money "transfer" transaction in a bank that transfers money from one account. We want to transfer $100 from account number 1001 to account number1005. If account 1001 does not have at least $100, the transaction does nothing to the database

Accounts
Table Before & After Transfer
(ActNum is the primary key)

Before transaction

AccNum	Firstname	Surname	Balance ($)
1001	Peace	Dadzie	700.00
1002	Grace	Pomaa	1000.00
1003	George	Antwi	400.00
1004	Papa	Kojo	800.00
1005	Doris	Sekyi	500.00

After Transaction

AccNum	Firstname	Surname	Balance ($)
1001	Peace	Dadzie	700.00
1002	Grace	Pomaa	1000.00
1003	George	Antwi	400.00
1004	Papa	Kojo	800.00
1005	Doris	Sekyi	600.00

First, the transaction needs to see if
there is at least $100 in the account.

```
START TRANSACTION;
SELECT AccNum, Balance
FROM Accounts
WHERE AccNum = 1001;
```

If the result is less than 100, then the transaction should be aborted by executing

```
ROLLBACK;
```

Otherwise, the transfer can proceed by executing the statements:

```
UPDATE Accounts
SET Balance = Balance - 100
WHERE AccNum = 1001;
UPDATE Accounts
SET Balance = Balance + 100
WHERE AccNum = 1005;
COMMIT;
```

2.3 Transaction Properties

The transfer transaction has three parts. It first checks the account balance. Then if the balance is less than $100, the transaction is aborted. Otherwise, the money is transferred and the transaction committed. ACID Guarantees: Atomicity: The amount withdrawn from one account and its deposit to the other either succeeds or fails but there is no partial execution.

Consistency: No constraints will be violated (only constraint is the primary key column AccNum which is not impacted by the transaction).

Isolation: The transfer transaction will correctly move $100 from one account to another even in the presence of other simultaneously executing transactions that may be interested in modifying the same accounts.

Durability: Once the transaction has successfully executed, its effects will be reflected in the database, i.e., they will become permanent.

2.4 Concurrency Control in Relational Databases

Concurrency control in Database management systems ensures that database transactions are performed concurrently without violating the data integrity of the respective databases.

Concurrency control deals with the issues involved with allowing multiple people simultaneous access to shared entities (objects, data records, or some other representation).

Thus, concurrency control is an essential element for correctness in any system where two database transactions or more can access the same data concurrently, e.g., virtually in any general-purpose database system.

To understand how to implement concurrency control within your system you must start by understanding the basics of collisions– you can either avoid them or detect and then resolve them. The next step is to understand transactions, which are collections of actions that potentially modify two or more entities.

2.4.1 Reason for Concurrency Control

If concurrent transactions are allowed in an uncontrolled manner, some unexpected result may occur. Here are some typical examples:

- Lost update problem: A second transaction writes a second value of a data-item (datum) on top of a first value written by a first concurrent transaction, and the first value is lost to other transactions running

concurrently which need, by their precedence, to read the first value. The transactions that have read the wrong value end with incorrect results.

- The dirty read problem: Transactions read a value written by a transaction that has been later aborted. This value disappears from the database upon abort, and should not have been read by any transaction ("dirty read"). The reading transactions end with incorrect results.
- The incorrect summary problem: While one transaction takes a summary over values of a repeated data-item, a second transaction updated some instances of that data-item. The resulting summary does not reflect a correct result for any (usually needed for correctness) precedence order between the two transactions (if one is executed before the other), but rather some random result, depending on the timing of the updates, and whether a certain update result has been included in the summary or not.

2.4.2 Concurrency control mechanisms

The main categories of concurrency control mechanisms are:

Optimistic - With multi-user systems it is quite common to be in a situation where collisions are infrequent. Although the two of us are working with *Customer* objects, you're working with the Wayne Miller object while I work with the John Berg object and therefore, we won't collide. When this is the case optimistic locking becomes a viable concurrency control strategy. The idea is that you accept the fact that collisions occur infrequently, and instead of trying to prevent them you simply choose to detect them and then resolve the collision when it does occur.

Pessimistic - Pessimistic locking is an approach where an entity is locked in the database for the entire time that it is in application memory (often in the form of an object). A lock either limits or prevents other users from working with the entity in the database. A write lock indicates that the holder of the lock intends to update the entity

and disallows anyone from reading, updating, or deleting the entity. A read lock indicates that the holder of the lock does not want the entity to change while the hold the lock, allowing others to read the entity but not update or delete it. The scope of a lock might be the entire database, a table, a collection of rows, or a single row. These types of locks are called database locks, table locks, page locks, and row locks respectively.

The advantages of pessimistic locking are that it is easy to implement and guarantees that your changes to the database are made consistently and safely. The primary disadvantage is that this approach isn't scalable. When a system has many users, or when the transactions involve a greater number of entities, or when transactions are long lived, then the chance of having to wait for a lock to be released increases. Therefore, this limits the practical number of simultaneous users that your system can support.

Overly optimistic - With the strategy you neither try to avoid nor detect collisions, assuming that they will never occur. This strategy is appropriate for single user systems, systems where the system of record is guaranteed to be accessed by only one user or system process at a time, or read-only tables. These situations do occur. It is important to recognize that this strategy is completely inappropriate for multi-user systems.

CHAPTER QUESTIONS

1. . Explain what is meant by a transaction
2. State and explain four transaction states.
3. What is Concurrency control?
4. What is meant by interleaved concurrent execution of database transactions in a multi-user system?
5. Discuss why concurrency control is needed
6. What does the acronym ACID stand for?
7. Discuss the various transaction ACID rules
8. Discuss the reasons for ensuring concurrency control in database transactions
9. Explain the following concurrency control mechanisms and state at least one scenario each in which a mechanism is applicable:
    i. Optimistic
    ii. Pessimistic
    iii. Overly optimistic
10. Compare and contrast the following concepts:
    i. Interleaved and simultaneous concurrency
    ii. Serial vs serialisable schedule
    iii. Shared vs exclusive lock
    iv. Basic vs conservative 2PL
    v. Wait-for vs wound-wait deadlock prevention protocol
    vi. Deadlock vs livelock

3. FORMAL QUERY LANGUAGES

There are two kinds of query languages – relational algebra and relational calculus.

Relational algebra; and Relational Calculus

Relational algebra is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either unary or binary. They accept relations as their input and yield relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations. The fundamental operations of relational algebra are as follows –

3.1 Relational Algebra

3.1.1 Select Operation (σ)

It selects tuples that satisfy the given predicate from a relation. Its notation is given as: σp(r) Where σ stands for selection predicate and r stands for relation. p is prepositional logic formula which may use connectors like and, or, and not. These terms may use relational operators like – =, ≠, ≥, < , >, ≤.

For example:

$\sigma_{subject = "database"}(\text{Books})$

Output – Selects tuples from books where subject is 'database'.

$\sigma_{\text{subject = "database" and price = "450"}}(\text{Books})$

Output – Selects tuples from books where subject is 'database' and 'price' is 450.

$\sigma_{\text{subject = "database" and price = "450" or year > "2010"}}(\text{Books})$

Output – Selects tuples from books where subject is 'database' and 'price' is 450 or those books published after 2010.

3.1.2 Project Operation (∏)

It projects column(s) that satisfy a given predicate. It is given as ∏A1, A2, An (r) Where A1, A2, An are attribute names of relation r. Duplicate rows are automatically eliminated, as relation is a set.

For example

$\prod_{subject,\ author}(Books)$

Selects and projects columns named as subject and author from the relation Books.

3.1.3 Union Operation (∪)

It performs binary union between two given relations and is defined as −

$r \cup s = \{ t \mid t \in r \text{ or } t \in s\}$

Notation − r U s

Where r and s are either database relations or relation result set (temporary relation). For a union operation to be valid, the following conditions must hold −

r, and s must have the same number of attributes. Attribute domains must be compatible. Duplicate tuples are automatically eliminated.

∏ author (Books) ∪ ∏ author (Articles)

Output − Projects the names of the authors who have either written a book or an article or both.

3.1.4 Set Difference (−)

The result of set difference operation is tuples, which are present in one relation but are not in the second relation.

Notation − r – s

Finds all the tuples that are present in r but not in s.

Example

$\prod_{author}(Books) - \prod_{author}(Articles)$

Output − Provides the name of authors who have written books but not articles.

3.1.5 Cartesian Product (X)

Combines information of two different relations into one.

Notation − r X s

Where r and s are relations and their output will be defined as:

$r \text{ X } s = \{ q\ t \mid q \in r \text{ and } t \in s\}$

Example

$\sigma_{author = 'Agangiba'}$(Books X Articles)

Output − Yields a relation, which shows all the books and articles written by Agangiba.

3.1.6 Rename Operation (ρ)

The results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. 'rename' operation is denoted with small Greek letter rho ρ.

Notation − ρ x (E)

Where the result of expression E is saved with name of x.

3.2 Relational Calculus

In contrast to Relational Algebra, Relational Calculus is a non-procedural query language, that is, it tells what to do but never explains how to do it. Relational calculus exists in two forms; namely Tuple Relational Calculus (TRC) and Domain Relational Calculus (DRC).

3.2.1 Tuple Relational Calculus (TRC)

Filtering variable ranges over tuples. Notation − {T | Condition}. Returns all tuples T that satisfies a condition.

For example

{T.name | Author(T) AND T.article = 'database'}

Output − Returns tuples with 'name' from Author who has written article on 'database'. TRC can be quantified. We can use Existential (∃) and Universal Quantifiers (∀).

For example

{R | ∃T ∈ Authors(T.article='database' AND R.name=T.name)}

Output − the above query will yield the same result as the previous one.

3.2.2 Domain Relational Calculus (DRC)

In DRC, the filtering variable uses the domain of attributes instead of entire tuple values (as done in TRC, mentioned above).

Notation – { a1, a2, a3, ..., an | P (a1, a2, a3, ... ,an)}

Where a1, a2 are attributes and P stands for formulae built by inner attributes.

For example

{< article, page, subject > | ∈ TutorialsPoint ∧ subject = 'database'}

Output – Yields Article, Page, and Subject from the relation TutorialsPoint, where subject is database. Just like TRC, DRC can also be written using existential and universal quantifiers. DRC also involves relational operators. The expression power of Tuple Relation Calculus and Domain Relation Calculus is equivalent to Relational Algebra.

CHAPTER QUESTIONS

Consider the following relational database schema consisting of the four relation schemas:

passenger (pid, pname, pgender, pcity)
agency (aid, aname, acity)
flight (fid, fdate, time, src, dest)
booking (pid, aid, fid, fdate)
Answer the following questions using relational algebra queries;

i. Get the complete details of all flights to London.
ii. Get the details about all flights from Ghana to London.
iii. Find only the flight numbers for passenger with pid 123 for flights to Ghana before 25/06/2022.
iv. Find the passenger names for passengers who have bookings on at least one flight.
v. Get the details of flights that are scheduled on both dates 01/2/2022 and 02/3/2022 at 16:00 hours.
vi. Find the details of all male passengers who are associated with Jet agency.

4. No SQL DATABASES

A NoSQL originally referring to non-SQL or non-relational is a database that provides a mechanism for storage and retrieval of data. This data is modeled in means other than the tabular relations used in relational databases. Such databases came into existence in the late 1960s, but did not obtain the NoSQL moniker until a surge of popularity in the early twenty-first century. NoSQL databases are used in real-time web applications and big data and their use are increasing over time. NoSQL systems are also sometimes called Not only SQL to emphasize the fact that they may support SQL-like query languages.

A NoSQL database includes simplicity of design, simpler horizontal scaling to clusters of machines and finer control over availability. The data structures used by NoSQL databases are different from those used by default in relational databases which makes some operations faster in NoSQL. The suitability of a given NoSQL database depends on the problem it should solve. Data structures used by NoSQL databases are sometimes also viewed as more flexible than relational database tables.

Many NoSQL stores compromise consistency in favor of availability, speed and partition tolerance. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages, lack of standardized interfaces, and huge previous investments in existing relational databases. Most NoSQL stores lack true ACID (Atomicity, Consistency, Isolation, Durability) transactions but a few databases, such as MarkLogic, Aerospike, FairCom c-treeACE, Google Spanner (though technically a NewSQL database), Symas LMDB, and OrientDB have made them central to their designs.

Most NoSQL databases offer a concept of eventual consistency in which database changes are propagated to all nodes so queries for data might not return updated data immediately or might result in reading data that is not accurate which is a problem known as stale reads. Also, some NoSQL systems may exhibit lost writes

and other forms of data loss. Some NoSQL systems provide concepts such as write-ahead logging to avoid data loss. For distributed transaction processing across multiple databases, data consistency is an even bigger challenge. This is difficult for both NoSQL and relational databases. Even current relational databases do not allow referential integrity constraints to span databases. There are few systems that maintain both X/Open XA standards and ACID transactions for distributed transaction processing.

Advantages of NoSQL:

There are many advantages of working with NoSQL databases such as MongoDB and Cassandra. The main advantages are high scalability and high availability.

High scalability –

NoSQL database use sharding for horizontal scaling. Partitioning of data and placing it on multiple machines in such a way that the order of the data is preserved is sharding. Vertical scaling means adding more resources to the existing machine whereas horizontal scaling means adding more machines to handle the data. Vertical scaling is not that easy to implement but horizontal scaling is easy to implement. Examples of horizontal scaling databases are MongoDB, Cassandra etc. NoSQL can handle huge amount of data because of scalability, as the data grows NoSQL scale itself to handle that data in efficient manner.

High availability –

Auto replication feature in NoSQL databases makes it highly available because in case of any failure data replicates itself to the previous consistent state.

Disadvantages of NoSQL:

NoSQL has the following disadvantages.

Narrow focus –

NoSQL databases have very narrow focus as it is mainly designed for storage but it provides very little functionality. Relational databases are a better choice in the field of Transaction Management than NoSQL.

Open-source –

NoSQL is open-source database. There is no reliable standard for NoSQL yet. In other words, two database systems are likely to be unequal.

Management challenge –

The purpose of big data tools is to make management of a large amount of data as simple as possible. But it is not so easy. Data management in NoSQL is much more complex than a relational database. NoSQL, in particular, has a reputation for being challenging to install and even more hectic to manage on a daily basis.

GUI is not available –

GUI mode tools to access the database is not flexibly available in the market.

Backup Backup is a great weak point for some NoSQL databases like MongoDB. MongoDB has no approach for the backup of data in a consistent manner.

Large document size –

Some database systems like MongoDB and CouchDB store data in JSON format. Which means that documents are quite large (BigData, network bandwidth, speed), and having descriptive key names actually hurts, since they increase the document size.

Types of NoSQL database:

Types of NoSQL databases and the name of the databases system that falls in that category are:

MongoDB falls in the category of NoSQL document-based database.

Key value store: Memcached, Redis, Coherence

Tabular: Hbase, Big Table, Accumulo

Document based: MongoDB, CouchDB, Cloudant

When should NoSQL be used:

When huge amount of data needs to be stored and retrieved.

The relationship between the data you store is not that important

The data changing over time and is not structured.

Support of Constraints and Joins is not required at database level

The data is growing continuously and you need to scale the database regular to handle the data.

5. XML DATABASES

XML Database is used to store huge amount of information in the XML format. As the use of XML is increasing in every field, it is required to have a secured place to store the XML documents. The data stored in the database can be queried using XQuery, serialized, and exported into a desired format.

5.1 XML Database Types

There are two major types of XML databases −XML- enabled and Native XML (NXD)

5.1.1 XML - Enabled Database

XML enabled database is nothing but the extension provided for the conversion of XML document. This is a relational database, where data is stored in tables consisting of rows and columns. The tables contain set of records, which in turn consist of fields.

5.1.2 Native XML Database

Native XML database is based on the container rather than table format. It can store large amount of XML document and data. Native XML database is queried by the **XPath**-expressions.

Native XML database has an advantage over the XML-enabled database. It is highly capable to store, query and maintain the XML document than XML-enabled database.

Example

Following example demonstrates XML database −

```
<?xml version = "1.0"?>
<contact-info>
  <contact1>
    <name>Tanmay Patil</name>
    <company>TutorialsPoint</company>
    <phone>(011) 123-4567</phone>
  </contact1>

  <contact2>
    <name>Manisha Patil</name>
    <company>TutorialsPoint</company>
    <phone>(011) 789-4567</phone>
  </contact2>
</contact-info>
```

Here, a table of contacts is created that holds the records of contacts (contact1 and contact2), which in turn consists of three entities − *name, company* and *phone*

CHAPTER QUESTIONS

1. What is NoSQL database?
2. Give four advantages of NoSQL
3. Describe the four types of NoSQL database.
4. Mention 3 feature in MongoDB
5. Give examples for NoSQL database?
6. What are some examples of NoSQL architecture?
7. What are the examples of wide column NoSQL database?
8. How does a NoSQL database work?
9. Briefly describe the advantages of mongodb database.
10. When is it appropriate to use NoSQL instead of SQL?