Database

Introduction to DBMS

Introduction to DBMS

- What is Database & DBMS?
- The need for a database
- The File-Based Systems
- Features of DBMS
- Usage of Database

Introduction to DBMS

- Database can be defined as the storage of interrelated data that has been organized in such a fashion that the process of retrieving data is effective and efficient

- **DBMS contains information about a particular enterprise**
  - Collection of interrelated data
  - Set of programs to access the data
  - An environment that is both *convenient* and *efficient* to use

- **Database Applications:**
  - Banking: all transactions
  - Airlines: reservations, schedules
  - Universities: registration, grades
  - Sales: customers, products, purchases
  - Online retailers: order tracking, customized recommendations

LTI
Let's Solve

Purpose of Database Systems

- In the early days, database applications were built directly on top of file systems
- Drawbacks of using file systems to store data:
  - Data redundancy and inconsistency
  - Difficulty in accessing data
  - Data isolation — multiple files and formats
  - Integrity problems

Purpose of Database Systems (Cont.)

- Drawbacks of using file systems (cont.)
  - Atomicity of updates
    - Failures may leave database in an inconsistent state with partial updates carried out
    - Example: Transfer of funds from one account to another should either complete or not happen at all
  - Concurrent access by multiple users
    - Concurrent accessed needed for performance
    - Uncontrolled concurrent accesses can lead to inconsistencies
      - Example: Two people reading a balance and updating it at the same time
  - Security problems
    - Hard to provide user access to some, but not all, data

- Database systems offer solutions to all the above problems

DBMS Architecture

DBMS Architecture

- Three-level architecture of DBMS
- The functions of Database Systems
- Overall system architecture

Levels of Architecture

- **Physical level:**

- Physical level describes the physical storage structure of data in database
- It is also known as Internal Level
- This level is very close to physical storage of data
- At lowest level, it is stored in the form of bits with the physical addresses on the secondary storage device
- At highest level, it can be viewed in the form of files
- The internal schema defines the various stored data types. It uses a physical data model

Levels of Architecture

- Logical/Conceptual level:

- Conceptual level describes the structure of the whole database for a group of users
- It is also called as the data model
- Conceptual schema is a representation of the entire content of the database
- These schema contains all the information to build relevant external records
- It hides the internal details of physical storage

Levels of Architecture

- **View/External level:** application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes

- External level is related to the data which is viewed by individual end users
- This level includes a no. of user views or external schemas
- This level is closest to the user
- External view describes the segment of the database that is required for a particular user group and hides the rest of the database from that user group
- At lowest level, it is stored in the form of bits with the physical addresses on the secondary storage device
- At highest level, it can be viewed in the form of files

Introduction to Data Modelling

LTI
Let's Solve

A Larsen & Toubro Group Company

Introduction to Data Modelling

- Explain the structure of Data
- Explain the process of data access in various data-models
- Explain the steps involved in the database designing pattern
- Design a Conceptual database using ER model

LTI
Let's Solve

Data Models

- According to Hoberman (2009),
  "A data model is a way of finding the tools for both business and IT professionals, which uses a set of symbols and text to precisely explain a subset of real information to improve communication within the organization and thereby lead to a more flexible and stable application environment"

  A data model is an idea which describes how the data can be represented and accessed from software system after its complete implementation
- It is a simple abstraction of complex real world data gathering environment
- It defines data elements and relationships among various data elements for a specified system
- The main purpose of data model is to give an idea that how final system or software will look like after development is completed

LTI
Let's Solve

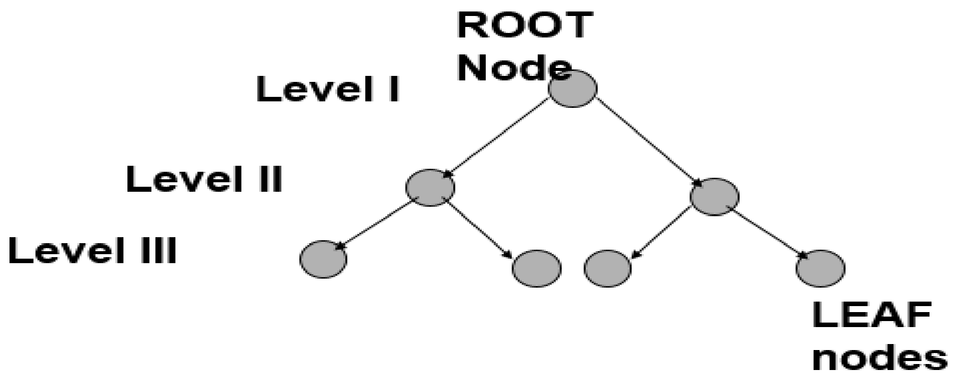
Data Models

Types

- Hierarchical DBMS
- Network DBMS
- Relational DBMS
- Object Relational DBMS
- Object Oriented DBMS

Hierarchical Data Model

- Definition
  - A hierarchical data model is a model that organizes data in a hierarchical tree structure
- Description
  - A hierarchical tree structure is made up of nodes and branches
  - The dependent nodes are at lower levels in the tree



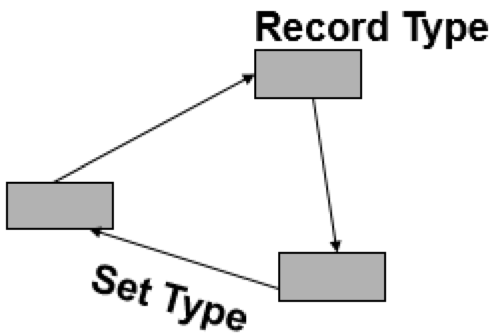
Network Data Model

Definition

- The network data model interconnects the entities of an enterprise into a network

Description

- A block represents an entity or record type. Each record type is composed of zero, one, or more attributes



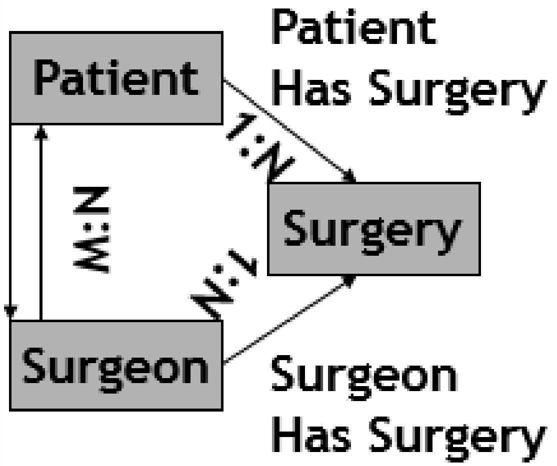
Network Data Model

1:N Relationship

- An owner record type owns zero, one, or many occurrences of a member record type

M:N Relationship

- A many-to-many relationship can be implemented by creating two one-to-many relationships
- Two record types are connected with a third entity type called connector record type
- In this case member record type has two owner record type



Relational model

- Example of tabular data in the relational model

Attributes

customer_id	*customer_name*	*customer_street*	*customer_city*	*account_number*
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-101
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-201
677-89-9011	Hayes	3 Main St.	Harrison	A-102
182-73-6091	Turner	123 Putnam St.	Stamford	A-305
321-12-3123	Jones	100 Main St.	Harrison	A-217
336-66-9999	Lindsay	175 Park Ave.	Pittsfield	A-222
019-28-3746	Smith	72 North St.	Rye	A-201

Database Design

The process of designing the general structure of the database:

- Logical Design – Deciding on the database schema. Database design requires that we find a “good” collection of relation schemas
  - Business decision – What attributes should we record in the database?
  - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
- Physical Design – Deciding on the physical layout of the database

Relational Model

- Structure of Relational Databases
- Fundamental Relational-Algebra-Operations
- Additional Relational-Algebra-Operations
- Extended Relational-Algebra-Operations
- Null Values
- Modification of the Database

Example of a Relation

account_number	*branch_name*	*balance*
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

LTI
Let's Solve

Relational Algebra

- Procedural language
- Six basic operators
  - select: σ
  - project: $\prod$
  - union: $\cup$
  - set difference: –
  - Cartesian product: x
  - rename: ρ
- The operators take one or two relations as inputs and produce a new relation as a result

Select Operation(σ) – Example

- Syntax: $\sigma_p(r)$

Where, σ represents the Select Predicate, r is the name of relation(table name in which you want to look for data), and p is the prepositional logic, where we specify the conditions that must be satisfied by the data

e.g:

$\sigma_{age > 17}$ **(Student)**

This will fetch the tuples(rows) from table Student, for which age will be greater than 17

σage > 17 and gender = 'Male' (Student)

This will return tuples(rows) from table Student with information of male students, of age more than 17.(Consider the Student table has an attribute Gender too)

Project Operation(∏) – Example

- Project operation is used to project only a certain set of attributes of a relation

Syntax: ∏A1, A2...(r)

where A1, A2 etc are attribute names(column names)

For example,

∏Name, Age(Student)

Above statement will show us only the Name and Age columns for all the rows of data in Student table

Union Operation(U) – Example

- This operation is used to fetch data from two relations(tables) or temporary relation(result of another operation)
- For this operation to work, the relations(tables) specified should have same number of attributes(columns) and same attribute domain. Also the duplicate tuples are automatically eliminated from the result

Syntax: A ∪ B

where A and B are relations.

For example, if we have two tables RegularClass and ExtraClass, both have a column student to save name of student, then,

∏Student(RegularClass) ∪ ∏Student(ExtraClass)

Above operation will give us name of Students who are attending both regular classes and extra classes, eliminating repetition

Intersection Operation(∩) – Example

- Defines a relation consisting of a set of all tuple that are in both A and B

Syntax: $A \cap B$

where A and B are relations

For example, if we want to find name of students who attend the regular class and the extra class as well, then, we can use the below operation:

$\prod_{Student}(RegularClass) \cap \prod_{Student}(ExtraClass)$

Set Difference Operation – Example

- This operation is used to find data present in one relation and not present in the second relation. This operation is also applicable on two relations, just like Union operation

Syntax: A - B

where A and B are relations

For example, if we want to find name of students who attend the regular class but not the extra class, then, we can use the below operation:

ΠStudent(RegularClass) - ΠStudent(ExtraClass)

Cartesian-Product Operation – Example

- Cross/Cartesian product is used to join two relations. For every row of Relation1, each row of Relation2 is concatenated. If Relation A has m tuples and and Relation B has n tuples, cross product of A and B will have m X n tuples

Syntax:

A X B

$\sigma_{\text{column 2}} = {}_{'1'} (A \times B)$

LTI
Let's Solve

Introduction to Data Modelling

- Quiz

In a relational model, relations are termed as?

- Tuples
- Attributes
- Tables
- Rows

Keys concepts in DBMS

Let's Solve

A Larsen & Toubro Group Company

Keys concepts in DBMS

- Super Key
- Primary Key
- Candidate Key
- Alternate Key
- Foreign Key
- Composite Key

Keys concepts

- Super Keys

Super Key is defined as a set of attributes within a table that can uniquely identify each record within a table. Super Key is a superset of Candidate key

- Primary Keys

A column or group of columns in a table which helps us to uniquely identifies every row in that table is called a primary key

- Candidate Keys

A super key with no redundant attribute is known as candidate key

- Composite Keys

A key that consists of more than one attribute to uniquely identify rows (also known as records & tuples) in a table is called composite key

Introduction to DBMS

- Alternate keys

All the keys which are not primary key are called an alternate key. It is a candidate key which is currently not the primary key

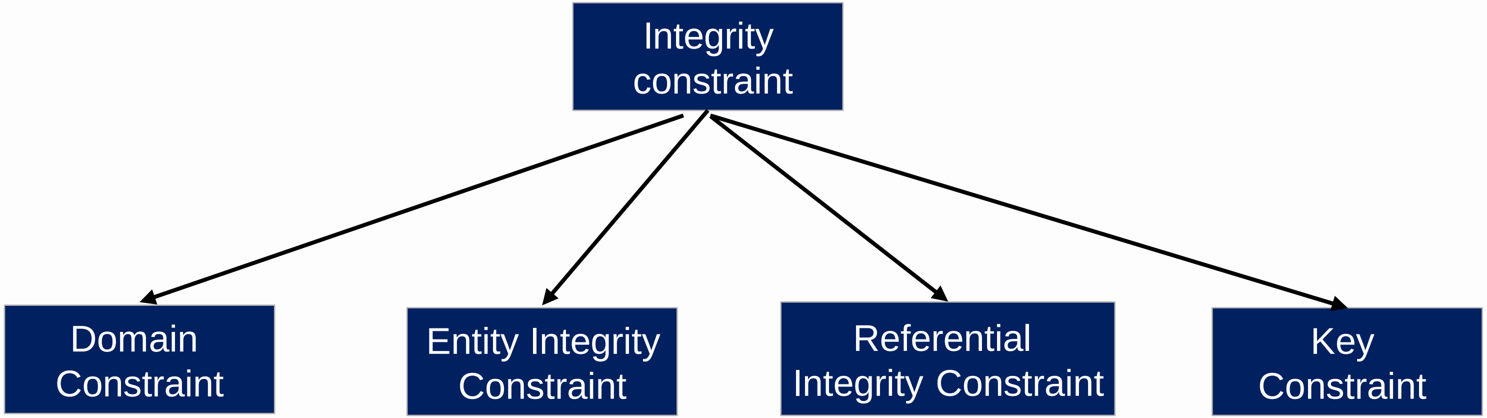
- Foreign Keys

Foreign keys are the columns of a table that points to the primary key of another table. They act as a cross-reference between tables

Constraints in DBMS

Integrity Constraint

- Integrity constraints are a set of rules. It is used to maintain the quality of information
- Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected
- Thus, integrity constraint is used to guard against accidental damage to the database



Constraints – Integrity types

Domain Integrity

- Domain constraints can be defined as the definition of a valid set of values for an attribute
- The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain

Entity integrity

- The entity integrity constraint states that primary key value can't be null
- This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows
- A table can contain a null value other than the primary key field

Constraints – Integrity types

Referential Integrity

- A referential integrity constraint is specified between two tables
- In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2

Which of the following is not a integrity constraint?

- Not null
- positive
- unique
- check

Introduction to Normalization

Normalization

- Explain the role of Normalization in database design
- Explain the steps in Normalization
- Types of Normal Forms – 1NF, 2NF, 3NF and Boyce Codd Normal Form(BCNF)

Normalization

- Normalization is the process of organizing the data in the database
- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies
- Normalization divides the larger table into the smaller table and links them using relationship
- The normal form is used to reduce redundancy from the database table

Anomalies

Relations that have redundant data may have problems called **anomalies**, which are classified as :

- Insertion anomalies
- Deletion anomalies
- Modification anomalies

STUDENT

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY	STUD_AGE
1	RAM	9716271721	Haryana	India	20
2	RAM	9898291281	Punjab	India	19
3	SUJIT	7898291981	Rajsthan	India	18
4	SURESH		Punjab	India	21

Table 1

STUDENT_COURSE

STUD_NO	COURSE_NO	COURSE_NAME
1	C1	DBMS
2	C2	Computer Networks
1	C2	Computer Networks

Table 2

LTI
Let's Solve

Insertion Anomalies

If a tuple is inserted in referencing relation and referencing attribute value is not present in referenced attribute, it will not allow inserting in referencing relation. For Example, If we try to insert a record in STUDENT_COURSE with STUD_NO =7, it will not allow

STUDENT

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY	STUD_AGE
1	RAM	9716271721	Haryana	India	20
2	RAM	9898291281	Punjab	India	19
3	SUJIT	7898291981	Rajsthan	India	18
4	SURESH		Punjab	India	21

Table 1

STUDENT_COURSE

STUD_NO	COURSE_NO	COURSE_NAME
1	C1	DBMS
2	C2	Computer Networks
1	C2	Computer Networks

Table 2

Updation & Deletion Anomalies

- If a tuple is deleted or updated from referenced relation and referenced attribute value is used by referencing attribute in referencing relation, it will not allow deleting the tuple from referenced relation. For Example, If we try to delete a record from STUDENT with STUD_NO =1, it will not allow. To avoid this, following can be used in query:

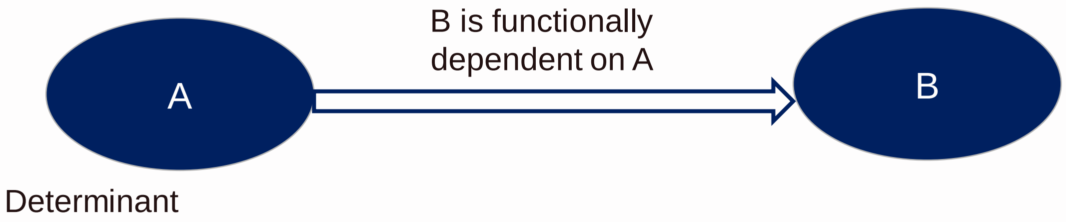
- ON DELETE/UPDATE SET NULL: If a tuple is deleted or updated from referenced relation and referenced attribute value is used by referencing attribute in referencing relation, it will delete/update the tuple from referenced relation and set the value of referenced attribute to NULL

- ON DELETE/UPDATE CASCADE: If a tuple is deleted or updated from referenced relation and referenced attribute value is used by referencing attribute in referencing relation, it will delete/update the tuple from referenced relation and referencing relation as well

Functional Dependencies

Functional dependency describes the relationship between attributes in a relation

For example, if A and B are attributes of relation R, and B is functionally dependent on A (denoted A B), if each value of A is associated with exactly one value of B. (A and B may each consist of one or more attributes)



Refers to the attribute or group of attributes on left hand side of the arrow of a functional dependency

Functional Dependencies(Contd.)

Trivial functional dependency:

- A → B has trivial functional dependency if B is a subset of A
- The following dependencies are also trivial like: A → A, B → B
- **Example:**
- Consider a table with two columns Employee_Id and Employee_Name

{Employee_id, Employee_Name} → Employee_Id is a trivial functional dependency as

Employee_Id is a subset of {Employee_Id, Employee_Name}

- Also, Employee_Id → Employee_Id and Employee_Name → Employee_Name are trivial dependencies too

LTI
Let's Solve

Functional Dependencies(Contd.)

Non-trival functional dependency:

- A → B has a non-trivial functional dependency if B is not a subset of A
- When A intersection B is NULL, then A → B is called as complete non-trivial
- **Example:**
- ID → Name,
- Name → DOB

Functional Dependencies(Contd.)

Transitive dependency:

- A transitive is a type of functional dependency which happens when t is indirectly formed by two functional dependencies

Example:

Company	CEO	Age
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Alibaba	Jack Ma	54

{Company} -> {CEO} (if we know the company, we know its CEO's name)

{CEO } -> {Age} If we know the CEO, we know the Age

Therefore according to the rule of rule of transitive dependency:

{ Company} -> {Age} should hold, that makes sense because if we know the company name, we can know his age

Functional Dependencies(Contd.)

Inference Rules

A set of all functional dependencies that are implied by a given set of functional dependencies X is called closure of X, written X+. A set of inference rule is needed to compute X+ from X

Armstrong's axioms

1. Reflexivity:	If B is a subset of A, them A → B
2. Augmentation:	If A → B, then A, C → B
3. Transitivity:	If A → B and B → C, then A → C
4. Self-determination:	A → A
5. Decomposition:	If A → B,C then A → B and A → C
6. Union:	If A → B and A → C, then A → B,C
7. Composition:	If A → B and C → D, then A,C → B,

Types of Normal Forms

Normal Form	Description
1NF	A relation is in 1NF if it contains an atomic value
2NF	A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key
3NF	A relation will be in 3NF if it is in 2NF and no transition dependency exists
4NF/BCNF	A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency

First Normal Form (1NF)

- A relation will be 1NF if it contains an atomic value
- It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute
- First normal form disallows the multi-valued attribute, composite attribute, and their combinations

First Normal Form (1NF)

- **Example:** Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP_PHONE

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385, 9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389, 8589830302	Punjab

LTI
Let's Solve

First Normal Form (1NF)

The decomposition of the EMPLOYEE table into 1NF has been shown below:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385	UP
14	John	9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389	Punjab
12	Sam	8589830302	Punjab

Second Normal Form (2NF)

- **Second normal form (2NF)** is a relation that is in first normal form and every non-primary-key attribute is fully functionally dependent on the primary key
- The normalization of 1NF relations to 2NF involves the removal of **partial dependencies**. If a partial dependency exists, we remove the function dependent attributes from the relation by placing them in a new relation along with a copy of their determinant

Second Normal Form (2NF)

For example:

Consider the table in which there are three below columns:

STUD_NO	COURSE_NO	COURSE_FEE
1	C1	1000
2	C2	1500
1	C4	2000
4	C3	1000
4	C1	1000
2	C5	2000

COURSE_NO -> COURSE_FEE , i.e., COURSE_FEE is dependent on COURSE_NO, which is a proper subset of the candidate key. Non-prime attribute COURSE_FEE is dependent on a proper subset of the candidate key, which is a partial dependency and so this relation is not in 2NF

Third Normal Form (3NF)

Transitive dependency

A condition where A, B, and C are attributes of a relation such that if A → B and B → C, then C is transitively dependent on A via B (provided that A is not functionally dependent on B or C)

Third normal form (3NF)

A relation that is in first and second normal form, and in which no non-primary-key attribute is transitively dependent on the primary key

The normalization of 2NF relations to 3NF involves the removal of transitive dependencies by placing the attribute(s) in a new relation along with a copy of the determinant

Third Normal Form (3NF)

Example: Suppose a company wants to store the complete address of each employee, they create a table named employee details that looks like this:

emp_id	emp_name	emp_zip	emp_state	emp_city	emp_district
1001	John	282005	UP	Agra	Dayal Bagh
1002	Ajeet	222008	TN	Chennai	M-City
1006	Lora	282007	TN	Chennai	Urrapakkam
1101	Lilly	292008	UK	Pauri	Bhagwan
1201	Steve	222999	MP	Gwalior	Ratan

Here, emp_state, emp_city & emp_district dependent on emp_zip. And, emp_zip is dependent on emp_id that makes non-prime attributes (emp_state, emp_city & emp_district) transitively dependent on super key (emp_id). This violates the rule of 3NF

Third Normal Form (3NF)

employee table:

emp_id	emp_name	emp_zip

employee_zip table:

emp_zip	emp_state	emp_city	emp_district

Tables in second normal form (2NF):?

Eliminate all hidden dependencies

Eliminate the possibility of an insertion anomalies

Have a composite key

Have all non key fields depend on the whole primary key

Let's Solve

Complete the following statement:

A relation is in ___________ if no non key attribute is transitively dependent on the primary key.

- 1NF
- 4NF
- 2NF
- 3NF

Let's Solve

Structured Query Language (SQL)

A Larsen & Toubro Group Company

SQL - Basic Operations

- Work with the SQL Data Definition Language (DDL)
- Work with the SQL Data Manipulation Language (DML)
- Write Queries using SQL select statements
- Work with SQL Operators
- Work with SQL Functions

Data Definition Language (DDL)

- Data Definition Language (DDL) is a standard for commands that define the different structures in a database
- DDL Statements are
- CREATE :Use to create objects like CREATE TABLE, CREATE FUNCTION, CREATE SYNONYM, CREATE VIEW. Etc.
- ALTER :Use to Alter Objects like ALTER TABLE, ALTER USER, ALTER DATABASE.
- DROP :Use to Drop Objects like DROP TABLE, DROP USER, DROP FUNCTION. Etc.
- TRUNCATE :Use to truncate (delete all rows) a table

Data Manipulation Language (DML)

- Data Manipulation Language is used to edit the data present in the database.DDL Statements are:
- INSERT- It is used to enter the data into the records of the table.
- UPDATE- It is used to update the records in the table
- DELETE- It is used to delete the existing records in the table.

Data Manipulation Language (DML)

- Data Manipulation Language is used to edit the data present in the database.DDL Statements are:
- INSERT- It is used to enter the data into the records of the table.
- UPDATE- It is used to update the records in the table
- DELETE- It is used to delete the existing records in the table.
- MERGE- Merging data from multiple tables.

LTI
Let's Solve

Data Query Language (DQL) and Data Control Language(DCL)

- Data Query Language is used to retrieve the data present in the database. DQL Statements are:
  - SELECT
- Data Control language is used to control access to the database. DCL statements are:
  - GRANT
  - REVOKE

Selecting data from columns

- SQL statement to display all the information of all employees.

```
select * from employee;
```

- SQL statement to display firstname all employees.

```
select first_name,last_name from employees;
```

- SQL statement to display derived data:

```
select first_name+' '+last_name as 'Name' from employees
```

```
select avg(salary) as average_salary from employees
```

- DISTINCT: get only distinct values in a specified column of a table

```
select distinct(city) from locations
```

Let's Solve

Filtering rows

- Where clause is used to get the rows from the table that satisfy one or more conditions

```
SELECT select_list FROM table_name WHERE search_condition;
```

- Retrieve all products with the category id 1

```
SELECT product_id, product_name, category_id, model_year, list_price FROM
production.products WHERE category_id = 1
```

- Retrieve all products category id is 1 and the model is 2018.

```
SELECT product_id, product_name, category_id, model_year, list_price FROM
production.products WHERE category_id = 1 AND model_year = 2018
```

LTI
Let's Solve

Filtering rows with logical operators

- Retrieve all products category id is 1 and the model is 2018.

```
SELECT product_id, product_name, category_id, model_year,
list_price FROM production.products WHERE category_id = 1 AND
model_year = 2018
```

- Get the product whose brand id is one or two and list price is larger than 1,000:

```
SELECT * FROM production.products WHERE (brand_id = 1 OR
brand_id = 2) AND list_price > 1000
```

- Find the products whose list price is less than 200 or greater than 6,000 :

```
SELECT product_name, list_price FROM production.products WHERE
list_price < 200 OR list_price > 6000
```

LTI
Let's Solve

Filtering rows Between and Not Between:

- Find the products whose list prices are between 1,899 and 1,999.99

```
SELECT product_id, product_name, category_id, model_year,
list_price FROM production.products WHERE list_price BETWEEN
1899.00 AND 1999.99
```

- Get the products whose list prices are in the range 149.99 and 199.99

```
SELECT product_id, product_name, list_price FROM
production.products WHERE list_price NOT BETWEEN 149.99 AND
199.99
```

LTI
Let's Solve

Filtering rows IN and Not IN:

- Find the products whose list price is one of the following values: 89.99, 109.99, and 159.99:

```
SELECT product_name, list_price FROM production.products WHERE
list_price IN (89.99, 109.99, 159.99)
```

- Find the products whose list prices are not one of the prices above:

```
SELECT product_name, list_price FROM production.products WHERE
list_price NOT IN (89.99, 109.99, 159.99)
```

LTI
Let's Solve

Filtering rows NULL and Not NULL:

- Find the customers who do not have phone number recorded in the customers table:

  ```
  SELECT customer_id, first_name, last_name, phone FROM
  sales.customers WHERE phone = NULL
  ```

  ```
  SELECT customer_id, first_name, last_name, phone FROM
  sales.customers WHERE phone IS NULL
  ```

- Find the customers who do have the phone information:

  ```
  SELECT customer_id, first_name, last_name, phone FROM
  sales.customers WHERE phone IS NOT NULL
  ```

Filtering rows using wild cards:

- Find products whose name contains the string “Cruiser”

```
SELECT product_id, product_name, category_id, model_year,
list_price FROM production.products WHERE product_name LIKE
'%Cruiser%’
```

- Find the customers whose last name starts with the letter z:

```
SELECT customer_id, first_name, last_name FROM sales.customers
WHERE last_name LIKE ‘z%’
```

- Find customers whose last name ends with the string “er”:

```
SELECT customer_id, first_name, last_name FROM sales.customers
WHERE last_name LIKE '%er’
```

LTI
Let's Solve

Filtering rows using wild cards:

- Find the customers where the second character is the letter u:

```
SELECT customer_id, first_name, last_name FROM sales.customers
WHERE last_name LIKE '_u%’
```

Find the customers where the first character in the last name is Y or Z:

```
SELECT customer_id, first_name, last_name FROM sales.customers
WHERE last_name LIKE '[YZ]%'
```

- Find the customers where the first character in the last name is the letter in the range A through C:

```
SELECT customer_id, first_name, last_name FROM sales.customers
WHERE last_name LIKE '[A-C]%'
```

LTI
Let's Solve

Filtering rows using wild cards:

- Find customers where the first character in the last name is not the letter in the range A through X:

```
SELECT customer_id, first_name, last_name FROM sales.customers
WHERE last_name LIKE '[^A-X]%'
```

- Find customers where the first character in the first name is not the letter A:

```
SELECT customer_id, first_name, last_name FROM sales.customers
WHERE first_name NOT LIKE 'A%'
```

LTI
Let's Solve

Filtering rows with TOP:

```sql
SELECT TOP 10 product_name, list_price FROM production.products
ORDER BY list_price DESC;

SELECT TOP 1 PERCENT product_name, list_price FROM
production.products ORDER BY list_price DESC;

SELECT TOP 3 WITH TIES product_name, list_price FROM
production.products ORDER BY list_price DESC;
```

LTI
Let's Solve

With SQL, how do you select all the records from a table named “Persons” where the value of the column “FirstName” ends with an “a”?

SELECT * FROM Persons WHERE FirstName=’a’

SELECT * FROM Persons WHERE FirstName LIKE ‘a%’

SELECT * FROM Persons WHERE FirstName LIKE ‘%a’

SELECT * FROM Persons WHERE FirstName=’%a%’

LTI
Let’s Solve

Sql Server Built-in functiontions:

- Number functions:
  - ABS — Returns the absolute value of a number
  - CEILING — Returns the smallest integer value that is >= a number
  - FLOOR — Returns the largest integer value that is <= to a number
  - PI — Returns the value of PI
  - POWER — Returns the value of a number raised to the power of another number
  - RAND — Returns a random number
  - ROUND — Rounds a number to a specified number of decimal places
  - SQRT — Returns the square root of a number
  - SQUARE — Returns the square of a number

Sql Server Built-in functiontions:

- Character functions:
  - ASCII Returns the ASCII value for the specific character
  - CHAR Returns the character based on the ASCII code
  - CHARINDEX Returns the position of a substring in a string
  - CONCAT Adds two or more strings together
  - LEFT Extracts a number of characters from a string (starting from left)
  - LEN Returns the length of a string
  - LOWER Converts a string to lower-case
  - LTRIM Removes leading spaces from a string
  - RTRIM Removes trailing spaces from a string
  - STR Returns a number as string
  - SUBSTRING Extracts some characters from a string

Sql Server Built-in functiontions:

- Date functions:
  - CURRENT_TIMESTAMP Returns the current date and time
  - DATEADD Adds a time/date interval to a date and then returns the date
  - DATEDIFF Returns the difference between two dates
  - DATENAME Returns a specified part of a date (as string)
  - DATEPART Returns a specified part of a date (as integer)
  - DAY Returns the day of the month for a specified date
  - GETDATE Returns the current database system date and time
  - ISDATE Checks an expression and returns 1 if it is a valid date, otherwise 0
  - MONTH Returns the month part for a specified date (a number from 1 to 12)
  - SYSDATETIME Returns the date and time of the SQL Server
  - YEAR Returns the year part for a specified date

Sql Server Built-in functiontions:

- Aggregate functions:
  - AVG — The AVG() aggregate function calculates the average of non-NULL values in a set.
  - COUNT — The COUNT() aggregate function returns the number of rows in a group, including rows with NULL values.
  - MAX — The MAX() aggregate function returns the highest value (maximum) in a set of non-NULL values.
  - MIN (minimum) — The MIN() aggregate function returns the lowest value in a set of non-NULL values.
  - SUM — The SUM() aggregate function returns the summation of all non-NULL values a set.

Sql Server Built-in functiontions:

- Aggregate functions:
  - AVG — The AVG() aggregate function calculates the average of non-NULL values in a set.
  - COUNT — The COUNT() aggregate function returns the number of rows in a group, including rows with NULL values.
  - MAX — The MAX() aggregate function returns the highest value (maximum) in a set of non-NULL values.
  - MIN (minimum) — The MIN() aggregate function returns the lowest value in a set of non-NULL values.
  - SUM — The SUM() aggregate function returns the summation of all non-NULL values a set.

Sql Server Built-in function:

- Aggregate functions:
- Find the average list price of all products in the products table:
- `SELECT AVG(list_price) avg_product_price FROM products`
- Find the number of products whose price is greater than 500:
- `SELECT COUNT(*) product_count FROM products WHERE list_price > 500`
- Find the highest list price of all products:
- `SELECT MAX(list_price) max_list_price FROM products;`
- Find the lowest list price of all products:
- `SELECT MIN(list_price) min_list_price FROM production.products;`
- Calculate the total stock by product:
- `SELECT SUM(quantity) stock_count FROM products`

Which SQL function is used to count the number of rows in a SQL query?

COUNT()

NUMBER()

SUM()

COUNT(*)

Order by clause:

- Display customer list by the first name in ascending order:

```
SELECT first_name, last_name FROM customers ORDER BY
first_name;
```

- Display customer list by the first name in descending order:

```
SELECT firstname, lastname FROM customers ORDER BY first_name
DESC;
```

- Retrieve first name, last name, and city of the customerssorted by the city first and then by the first name.

```
SELECT city, first_name, last_name FROM customers ORDER BY
city, first_name;
```

Order by clause:

- Sort the customers by the city in descending order and the sort the sorted result set by the first name in ascending order.

```
SELECT city, first_name, last_name FROM sales.customers ORDER BY city
DESC, first_name ASC;
```

- Sort the customer by the state even though the state column does not appear on the select list.

```
SELECT city, first_name, last_name FROM customers ORDER BY state;
```

- Retrieve a customer list sorted by the length of the first name.

```
SELECT first_name, last_name FROM sales.customers ORDER BY LEN(first_name)
DESC;
```

LTI
Let's Solve

Group by clause:

- The GROUP BY clause allows you to arrange the rows of a query in groups. The groups are determined by the columns that you specify in the GROUP BY clause.
- Retrieve customer id and the year of the customers with the customer id one and two.
- `SELECT customer_id, YEAR (order_date) order_year FROM orders WHERE customer_id IN (1, 2) GROUP BY customer_id ORDER BY customer_id;`

- Retrieve customer id and the ordered year of the customers with the customer id one and two, the number of orders placed by the customer by year:
- `SELECT customer_id, YEAR (order_date) order_year FROM orders WHERE customer_id IN (1, 2) GROUP BY customer_id, YEAR (order_date) ORDER BY customer_id;`

Group by clause:

- Return the number of customers in every city:

```
SELECT city, COUNT (customer_id) customer_count FROM
sales.customers GROUP BY city ORDER BY city;
```

- Return number of customers by state and city:

```
SELECT city, state, COUNT (customer_id) customer_count FROM
sales.customers GROUP BY state, city ORDER BY city, state;
```

Group by .. having clause:

- The HAVING clause is often used with the GROUP BY clause to filter groups based on a specified list of conditions.
  - Find the customers who placed at least two orders per year:

```
SELECT customer_id, YEAR (order_date), COUNT (order_id) order_count
FROM
sales.orders
GROUP BY
customer_id,
YEAR (order_date)
HAVING
COUNT (order_id) >= 2
ORDER BY
customer_id;
```

LTI
Let's Solve

Group by .. having clause:

- Find sales orders whose net values are greater than 20,000:

```
SELECT
order_id,
SUM (
quantity * list_price * (1 - discount)
) net_value
FROM
sales.order_items
GROUP BY
order_id HAVING
SUM (
quantity * list_price * (1 - discount)
) > 20000 ORDER BY net_value;
```

-

LTI
Let's Solve

Group by .. having clause:

- Filter out the category which has the maximum list price greater than 4,000 or the minimum list price less than 500:

```
SELECT
category_id, MAX (list_price) max_list_price, MIN (list_price)
min_list_price
FROM
production.products
GROUP BY category_id
HAVING MAX (list_price) > 4000 OR MIN (list_price) < 500;
```

Group by .. having clause:

- Find product categories whose average list prices are between 500 and 1,000:

```
SELECT
category_id, AVG (list_price) avg_list_price
FROM production.products
GROUP BY category_id
HAVING AVG (list_price) BETWEEN 500 AND 1000;
```

LTI
Let's Solve

What is the meaning of “HAVING” clause in Sql?

- To filter out the row values
- To filter out the column values
- To filter out the row and column values
- None of the mentioned

Joins:

- In a relational database, data is distributed in multiple logical tables.
- To get a complete meaningful set of data, you need to query data from these tables by using joins.
- SQL Server supports many kinds of joins:
  - Inner join
  - Left join
  - Right join
  - Full outer join and
  - Cross join.
- Each join type specifies how SQL Server uses data from one table to select rows in another table.

Joins:

- Inner join:
  - produces a data set that includes rows from the left table which have matching rows from the right table.
  - Get the rows from the candidates table that have the corresponding rows with the same values in the fullname column of the employees table:

```
SELECT
c.id candidate_id, c.fullname candidate_name, e.id employee_id,
e.fullname employee_name
FROM
hr.candidates c
INNER JOIN
hr.employees e
ON e.fullname = c.fullname;
```

Joins:

- Left join:
  - Selects data starting from the left table and matching rows in the right table.
  - The left join returns all rows from the left table and the matching rows from the right table.
  - If a row in the left table does not have a matching row in the right table, the columns of the right table will have nulls.

```
SELECT
c.id candidate_id, c.fullname candidate_name, e.id
employee_id, e.fullname employee_name
FROM
hr.candidates c
LEFT JOIN
hr.employees e
ON e.fullname = c.fullname;
```

Joins:

- Right join:
- selects data starting from the right table. It is a reversed version of the left join.
- The right join returns a result set that contains all rows from the right table and the matching rows in the left table.
- If a row in the right table that does not have a matching row in the left table, all columns in the left table will contain nulls.

```
SELECT
c.id candidate_id, c.fullname candidate_name, e.id
employee_id, e.fullname employee_name
FROM hr.candidates c
RIGHT JOIN hr.employees e ON e.fullname = c.fullname;
```

LTI
Let's Solve

Joins:

- Full join:
  - Returns a result set that contains all rows from both left and right tables, with the matching rows from both sides where available.
  - In case there is no match, the missing side will have NULL values.

```
SELECT
c.id candidate_id, c.fullname candidate_name, e.id
employee_id, e.fullname employee_name
FROM
hr.candidates c
FULL JOIN
hr.employees e
ON e.fullname = c.fullname;
```

Joins:

- **Self join:**
- A self join allows you to join a table to itself.
- It is useful for querying hierarchical data or comparing rows within the same table.
- A self join uses the inner join or left join clause.
- Because the query that uses self join references the same table, the table alias is used to assign different names to the same table within the query.

```
SELECT
e.first_name + ' ' + e.last_name employee, m.first_name + ' '
+ m.last_name manager
FROM sales.staffs e
INNER JOIN sales.staffs m ON m.staff_id = e.manager_id ORDER
BY manager;
```

Joins:

- Self join:

- Find customers who locate in the same city.

```
SELECT
c1.city, c1.first_name + ' ' + c1.last_name customer_1,
c2.first_name + ' ' + c2.last_name customer_2
FROM sales.customers c1
INNER JOIN sales.customers c2
ON c1.customer_id > c2.customer_id AND c1.city = c2.city
ORDER BY city, customer_1, customer_2;
```

Joins:

- Cross join:
  - The CROSS JOIN joined every row from the first table (T1) with every row from the second table (T2). In other words, the cross join returns a Cartesian product of rows from both tables.
  - Return combinations of all products and stores:

```
SELECT
product_id, product_name, store_id, 0 AS quantity
FROM production.products
CROSS JOIN sales.stores
ORDER BY product_name, store_id;
```

LTI
Let's Solve

Which product is returned in a join query have no join condition?

- Equijoins
- Cartesian
- Both Equijoins and Cartesian
- None of the mentioned

LTI
Let's Solve

Subquery:

- A subquery is a query nested inside another statement:

```
SELECT order_id, order_date, customer_id
FROM sales.orders
WHERE customer_id
IN ( SELECT customer_id FROM sales.customers WHERE city =
'New York')
ORDER BY order_date DESC;
```

Subquery:

- SQL Server supports up to 32 levels of nesting.

```
SELECT product_name, list_price
FROM production.products
WHERE
list_price >
( SELECT AVG (list_price) FROM production.products WHERE brand_id
IN
( SELECT brand_id FROM production.brands WHERE brand_name = 'Strider'  OR
brand_name = 'Trek' ) )
ORDER BY list_price;
```

LTI
Let's Solve

Subquery IN operator:

```
SELECT
product_id, product_name
FROM production.products
WHERE category_id
IN
( SELECT category_id FROM production.categories WHERE category_name =
'Mountain Bikes' OR category_name = 'Road Bikes' );
```

LTI
Let's Solve

Subquery ANY operator:

```
SELECT
product_name, list_price
FROM production.products
WHERE list_price >=
ANY
( SELECT AVG (list_price) FROM production.products GROUP BY brand_id )
```

Subquery ALL operator:

```sql
SELECT
product_name, list_price
FROM production.products
WHERE list_price >=
ALL
( SELECT AVG (list_price) FROM production.products GROUP BY brand_id )
```

Let's Solve

Subquery EXISTS and NOT EXISTS operator:

```
 SELECT
customer_id, first_name, last_name, city
FROM sales.customers c
WHERE
EXISTS
( SELECT customer_id FROM sales.orders o WHERE o.customer_id
=  c.customer_id AND YEAR (order_date) = 2017 )
ORDER BY first_name, last_name;
```

LTI
Let's Solve

Subquery EXISTS and NOT EXISTS operator:

```
 SELECT
customer_id, first_name, last_name, city
FROM sales.customers c
WHERE
NOT EXISTS
( SELECT customer_id FROM sales.orders o WHERE o.customer_id
=  c.customer_id AND YEAR (order_date) = 2017 )
ORDER BY first_name, last_name;
```

Which of the following statement(s) is TRUE regarding subqueries?

- Inner queries in WHERE clause can contain ORDER BY
- Outer query and inner query can get data from different tables
- Outer query and inner query must get data from the same table
- Inner queries cannot contain GROUP BY clause

Data Definition Language:

```
CREATE DATABASE database_name;
DROP DATABASE [ IF EXISTS ] database_name [,database_name2,...];
CREATE TABLE sales.visits (
visit_id INT PRIMARY KEY IDENTITY (1, 1),
first_name VARCHAR (50) NOT NULL,
last_name VARCHAR (50) NOT NULL,
visited_at DATETIME,
phone VARCHAR(20),
store_id INT NOT NULL,
FOREIGN KEY (store_id) REFERENCES sales.stores (store_id)
);
```

Data Definition Language:

```
ALTER TABLE table_name ADD column_name data_type column_constraint;
ALTER TABLE table_name ALTER COLUMN column_name new_data_type(size);
ALTER TABLE table_name DROP COLUMN column_name;
ALTER TABLE persons ADD full_name AS (first_name + ' ' + last_name);
EXEC sp_rename 'old_table_name', 'new_table_name'
DROP TABLE [IF EXISTS] [database_name.][schema_name.]table_name;
TRUNCATE TABLE [database_name.][schema_name.]table_name;
SELECT select_list INTO destination FROM source [WHERE condition]
```

LTI
Let's Solve

Constraints:

```
CREATE TABLE table_name ( pk_column data_type PRIMARY KEY, ... );

CREATE TABLE table_name ( pk_column_1 data_type, pk_column_2 data type,
... PRIMARY KEY (pk_column_1, pk_column_2) );

CONSTRAINT fk_constraint_name FOREIGN KEY (column_1, column2,...)
REFERENCES parent_table_name(column1,column2,..)

FOREIGN KEY (foreign_key_columns) REFERENCES
parent_table(parent_key_columns) ON UPDATE action ON DELETE action;

CHECK(condition)

ALTER TABLE table_name ADD CONSTRAINT constraint_name CHECK(condtion);

DROP CONSTRAINT constraint_name;

ALTER TABLE table_name NOCHECK CONSTRAINT constraint_name;
```

Constraints:

```
CONSTRAINT constraint_name UNIQUE(column_name)

UNIQUE (column1,column2)

ALTER TABLE table_name ADD CONSTRAINT constraint_name UNIQUE(column1,
column2,...);

ALTER TABLE table_name ALTER COLUMN column_name data_type NOT NULL;

ALTER TABLE table_name ALTER COLUMN column_name data_type NULL;

ALTER TABLE table_name ALTER COLUMN column_name data_type default value;
```

LTI
Let's Solve

Data Manipulation Language:

```
INSERT INTO table_name (column_list) VALUES (value_list);

INSERT INTO table_name( column_list) OUTPUT inserted.column_name
VALUES value_list);

INSERT INTO table_name( column_list)

OUTPUT

inserted.column_name1,

inserted.column_name2

VALUES value_list);
```

-

Data Manipulation Language:

```
INSERT INTO table_name (column_list) VALUES (value_list_1), (value_list_2), ... (value_list_n);

INSERT [ TOP ( expression ) [ PERCENT ] ] INTO target_table (column_list) query

INSERT INTO table_name (column_list) SELECT column_list FROM table1 WHERE condition

INSERT TOP (n) INTO table_name (column_list) SELECT column_list FROM table_name ORDER BY column_name

UPDATE table_name SET c1 = v1, c2 = v2, ... cn = vn [WHERE condition]

DELETE [ TOP ( expression ) [ PERCENT ] ] FROM table_name [WHERE search_condition];

MERGE target_table USING source_table ON merge_condition WHEN MATCHED THEN update_statement WHEN NOT MATCHED THEN insert_statement WHEN NOT MATCHED BY SOURCE THEN DELETE;
```

Which of the following is/are the DDL statements?

- Create
- Drop
- Alter
- All of the Mentioned

Transaction Management

- What is Transaction?
  - A transaction is a unit of work that is performed against a database.
  - This work can be performed manually, such as an UPDATE statement you issue in SQL Server Management Studio or an application that INSERTS data into the database.
  - These are all transactions.

LTI
Let's Solve

Transaction Management

- SQL Server supports the following transaction modes:
  - **Autocommit transactions** - Each individual statement is a transaction.
  - **Explicit transactions** - Each transaction is explicitly started with the BEGIN TRANSACTION statement and explicitly ended with a COMMIT or ROLLBACK statement.
  - **Implicit transactions** – A new transaction is implicitly started when the prior transaction completes, but each transaction is explicitly completed with a COMMIT or ROLLBACK statement.

Transaction Management

- ACID properties:
- **Atomicity** - ensures that all operations within the work unit are completed successfully, otherwise the transaction is aborted at the point of failure and previous operations are rolled back to their former state.
- **Consistency** - ensures that the database properly changes states upon a successfully committed transaction.
- **Isolation** – enables transactions to operate independently of and transparent to each other.
- **Durability** - ensures that the result or effect of a committed transaction persists in case of a system failure.

Transaction Management

- SQL Server supports transaction control commands:
  - BEGIN TRANSACTION - the starting point of a transaction
  - ROLLBACK TRANSACTION - roll back a transaction either because of a mistake or a failure
  - COMMIT TRANSACTION - save changes to the database

Transaction Management

```
DECLARE @intErrorCode INT
BEGIN TRAN
UPDATE Authors
SET Phone = '415 354-9866'
WHERE au_id = '724-80-9391'
SELECT @intErrorCode = @@ERROR
IF (@intErrorCode <> 0) GOTO PROBLEM
UPDATE Publishers
SET city = 'Calcutta', country = 'India'
WHERE pub_id = '9999'
SELECT @intErrorCode = @@ERROR
IF (@intErrorCode <> 0) GOTO PROBLEM
COMMIT TRAN

PROBLEM:
IF (@intErrorCode <> 0) BEGIN
PRINT 'Unexpected error occurred!'
ROLLBACK TRAN
END
```

Let's Solve

Transaction Management

```
SELECT 'Before BEGIN TRAN', @@TRANCOUNT -- The value of @@TRANCOUNT is 0
BEGIN TRAN
SELECT 'After BEGIN TRAN', @@TRANCOUNT -- The value of @@TRANCOUNT is 1
DELETE sales
BEGIN TRAN nested
SELECT 'After BEGIN TRAN nested', @@TRANCOUNT
-- The value of @@TRANCOUNT is 2
DELETE titleauthor
COMMIT TRAN nested
-- Does nothing except decrement the value of @@TRANCOUNT

SELECT 'After COMMIT TRAN nested', @@TRANCOUNT
-- The value of @@TRANCOUNT is 1
ROLLBACK TRAN

SELECT 'After ROLLBACK TRAN', @@TRANCOUNT -- The value of @@TRANCOUNT is 0
-- because ROLLBACK TRAN always rolls back all transactions and sets
-- @@TRANCOUNT to 0
```

___________ marks the end of a successful implicit or explicit transaction.

COMMIT TRANSACTION

ROLLBACK TRANSACTION

COMMIT WORK

All of the mentioned

Views

- A view is a named query stored in the database catalog that allows you to refer to it later.
- A view may consist of columns from multiple tables using joins or just a subset of columns of a single table.
- Views is useful for abstracting or hiding complex queries.
- DML operations can not be performed if view contains multiple tables data(using joins).
- **With Check Option:** WITH CHECK OPTION will make sure that all INSERT and UPDATE statements executed against the view meet the restrictions in the WHERE clause, and that the modified data in the view remains visible after INSERT and UPDATE statements.

```
CREATE VIEW [OR ALTER] schema_name.view_name [(column_list)] AS
select_statement;
```

Views

```
CREATE VIEW customerInfo_View
AS
Select CustID,
FNAME,
LASTNME,
UserID
FROM dbo.Customer


select * from customerInfo_View
```

LTI
Let's Solve

Views

```
CREATE VIEW sales.daily_sales
AS
SELECT
year(order_date) AS y,
month(order_date) AS m,
day(order_date) AS d,
p.product_id,
product_name,
quantity * i.list_price AS sales
FROM sales.orders AS o
INNER JOIN sales.order_items AS i
ON o.order_id = i.order_id
INNER JOIN production.products AS p ON p.product_id =
i.product_id;
```

LTI
Let's Solve

Views

- Renaming a View:

```
EXEC sp_rename ‘old name’,’new name’;
```

- Drop a View:

```
DROP VIEW [IF EXISTS] schema_name.view_name;
```

- List all the views in the database:

```
select schema_name(schema_id) as schema_name, name as view_name
from sys.views order by schema_name, view_name
```

Syntax for creating views is __________

CREATE VIEW AS SELECT

CREATE VIEW AS UPDATE

DROP VIEW AS SELECT

CREATE VIEW AS UPDATE

T-SQL programming

- Transact-SQL is a database procedural programming language.
- Procedural languages are designed to extend SQL's abilities while being able to integrate well with SQL.
- T-SQL is organized by each block of statement.
- A block of statement can embrace another block of statement in it.
- A block of statement starts by BEGIN and finishes by END.

```
BEGIN
-- Declare variables
-- T-SQL Statements
END;
```

T-SQL programming

```
Begin
Declare @v_Result Int;
Declare @v_a Int = 50;
Declare @v_b Int = 100;
Print 'v_a= ' + Cast(@v_a as varchar(15));
Print 'v_b= ' + Cast(@v_b as varchar(15));
Set @v_Result = @v_a + @v_b;
Print 'v_Result= ' + Cast(@v_Result as varchar(15));
END
```

T-SQL programming

If… Else:

```
IF <condition 1> THEN
Job 1;
[ELSIF <condition 2> THEN
Job 2;
]
[ELSE
Job n + 1;
]
END IF;
```

LTI
Let's Solve

T-SQL programming

While Loop:

```
WHILE condition
BEGIN
{...statements...}
END;
```

T-SQL programming

Case .. When:

```
 CASE
WHEN condition1 THEN result1
WHEN condition2 THEN result2
WHEN conditionN THEN resultN
ELSE result
END;
```

The BEGIN and END statements are used when ______________

A WHILE loop needs to include a block of statements

An element of a CASE expression needs to include a block of statements

An IF or ELSE clause needs to include a block of statements

All of the mentioned

LTI
Let's Solve

Stored Procedures

- SQL Server stored procedures are used to group one or more Transact-SQL statements into logical units.
- The stored procedure are stored as named objects in the SQL Server Database Server.
- When you call a stored procedure for the first time, SQL Server creates an execution plan and stores it in the cache.
- In the subsequent executions of the stored procedure, SQL Server reuses the plan so that the stored procedure can execute very fast with reliable performance.

Stored Procedures

```
CREATE PROCEDURE procedure_name
AS
BEGIN
SQL Statement1;
SQL Statement2;
SQL Statement3;
SQL Statement4;
END;

EXEC procedure_name;
```

Stored Procedures

```sql
CREATE PROCEDURE uspProductList
AS
BEGIN
SELECT product_name, list_price FROM production.products
ORDER BY product_name;
END;
```

Stored Procedures

- Modifying a procedure:

```
ALTER PROCEDURE uspProductList

AS

BEGIN

SELECT product_name, list_price FROM production.products ORDER BY
list_price

END;
```

- Dropping a procedure:

```
DROP PROCEDURE sp_name;
```

LTI
Let's Solve

Stored Procedures with parameters:

```
CREATE PROCEDURE uspFindProducts(@min_list_price AS DECIMAL)
AS
BEGIN
SELECT product_name, list_price FROM production.products
WHERE list_price >= @min_list_price
ORDER BY list_price;
END;


EXECUTE uspFindProducts 900
```

LTI
Let's Solve

Stored Procedures with parameters:

```
ALTER PROCEDURE uspFindProducts( @min_list_price AS DECIMAL
,@max_list_price AS DECIMAL )

AS

BEGIN

SELECT product_name, list_price FROM production.products

WHERE list_price >= @min_list_price AND list_price <=
@max_list_price

ORDER BY list_price;

END;


EXECUTE uspFindProducts 900, 1000;
```

Stored Procedures with OUTPUT parameters:

```sql
CREATE PROCEDURE uspFindProductByModel ( @model_year SMALLINT,
@product_count INT OUTPUT )
AS
BEGIN
SELECT product_name, list_price FROM production.products
WHERE model_year = @model_year;
SELECT @product_count = @@ROWCOUNT;
END;
```

Stored Procedures with OUTPUT parameters:

- Calling stored procedures with output parameters:

```
DECLARE @count INT;
EXEC uspFindProductByModel @model_year = 2018, @product_count =
@count OUTPUT;
SELECT @count AS 'Number of products found';
```

A stored procedure in SQL is a ___________

- block of functions
- group of Transact-SQL statements compiled into a single execution plan.
- group of distinct SQL statements.
- None of the mentioned

Implementing functions:

- The SQL Server user-defined functions help you simplify your development by encapsulating complex business logic and make them available for reuse in every query.
- In SQL Server, user defined functions are of three types:
  - Scalar functions
  - Inline table-valued functions
  - Multi-statement table-valued functions

Implementing functions:

- Scalar functions:
  - Scalar functions takes one or more parameters and returns a single value.

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

Implementing functions:

- Scalar functions:

```
create function fnGetEmpFullName
(
@FirstName varchar(50),
@LastName varchar(50)
)
returns varchar(101)
As
Begin
return (Select @FirstName + ' '+ @LastName);
end
```

Implementing functions:

- Inline table-valued functions:
  - The user-defined inline table-valued function returns a table variable as a result of actions performed by the function.

```
create function fnGetEmployee()
returns Table
as
return (select * from Employee)
```

Implementing functions:

- Multi-statement table-valued functions:
  - A user-defined multi-statement table-valued function returns a table variable as a result of actions performed by the function.
  - In this, a table variable must be explicitly declared and defined whose value can be derived from multiple SQL statements.

Implementing functions:

- Multi-statement table-valued functions:

```
Create function fnGetMulEmployee()
returns @Emp Table
(
EmpID int,
FirstName varchar(50),
Salary int
)
As
begin
Insert into @Emp Select e.EmpID,e.FirstName,e.Salary from Employee e;
--Now update salary of first employee
update @Emp set Salary=25000 where EmpID=1;
--It will update only in @Emp table not in Original Employee table
return
end
```

Which of the following is not a User defined function?

Max()

Scalar Function

Inline Table-Valued Function

Multi-Statement Table-Valued Function

SQL Server Triggers

- SQL Server triggers are special stored procedures that are executed automatically in response to the database object, database, and server events.
- SQL Server provides three type of triggers:
  - Data manipulation language (DML) triggers which are invoked automatically in response to INSERT, UPDATE, and DELETE events against tables.
  - Data definition language (DDL) triggers which fire in response to CREATE, ALTER, and DROP statements. DDL triggers also fire in response to some system stored procedures that perform DDL-like operations.
  - Logon triggers which fire in response to LOGON events

SQL Server Triggers

- Create a trigger:
  - The CREATE TRIGGER statement allows you to create a new trigger that is fired automatically whenever an event such as INSERT, DELETE, or UPDATE occurs against a table.

```
CREATE TRIGGER [schema_name.]trigger_name
ON table_name
AFTER {[INSERT],[UPDATE],[DELETE]}
[NOT FOR REPLICATION]
AS
{sql_statements}
```

SQL Server Triggers

- Enabling and disabling a trigger:
  - To enable trigger:

```
ENABLE TRIGGER [schema_name.][trigger_name] ON [object_name |
DATABASE | ALL SERVER]
```

  - To disable trigger:

```
DISABLE TRIGGER [schema_name.][trigger_name] ON [object_name |
DATABASE | ALL SERVER]
```

  - To drop trigger:

```
DROP TRIGGER [ IF EXISTS ] [schema_name.]trigger_name [ ,...n ];
```

Structured Query Language (SQL)

- Quiz

The OLD and NEW qualifiers can be used in which type of trigger?

ROWLEVEL DML TRIGGERS

STATEMENT LEVEL DML TRIGGERS

ROW LEVEL SYSTEM TRIGGERS

STATEMENT LEVEL DML TRIGGERS

LTI
Let's Solve

LTI

Let's Solve