|  |  |  |  |
| --- | --- | --- | --- |
| No. | Title | Points to be covered / example | Length |
| 1 | Introduction to data types. | Note: Welcome to the session on Introduction to data types which is basic but very important topic to design databases.  What we going to cover in this session/ In this session we are going to cover/  Today’s topic are / Today we are going to discuss/.  Categories of datatype Numeric, string, binary  Discussion on numeric data types (int, smallint, etc) including their ranges.  Difference between dec andfloat.  Discussion on String datatype (char, varchar, text, etc) nonUnicode  Diff between char and varchar  Discussion on nvarchar/ Unicode  Diff between varchar and nvarchar (2)  Discussion on BLOB (full form to be told) – small, bigblob, etc.  Tell them the simillraty between blob and string.  CHAR, VARCHAR, TINYINT, INT, YEAR, DATE, BOOLEAN  Char(10)  Varchar(10)  e.g. todo  e.g.  DECIMAL(7,2) Total length of this data type is 7 of which 2 will be decimal part.  5396.17 ok value  5396.174 5396.17  5396.178 5396.18  5396.1743 5396.17  5396.1748 5396.17  5396.1788 5396.18  Note: With this we have covered all the today’s topics.  I hope you understood data types in SQL. | 15 |
| 2(P1) | SQL Commands | Note: Welcome back to the session on SQL Commands.  What are SQL commands   1. DDL – Data Definition Language 2. DML – Data Manipulation Language 3. DCL – Data Control Language 4. TCL – Transaction Control Language 5. Constraints   DDL – Discuss – Full list of commands with brief description  e.g. of DDL  **CREATE** TABLE student (  ID INT,  firstName VARCHAR(45),  lastName VARCHAR(45),  DoB DATE  );  **ALTER** TABLE student ADD COLUMN emailID VARCHAR(145);  **DROP** TABLE student;  DML– Discuss – Full list of commands with brief description  e.g. of DML  **INSERT** INTO student VALUES (1, 'raju', 'patel', '1970-12-10', 'raju123@gmail.com');  **SELECT** \* FROM student;  **UPDATE** student SET emailID = 'mohan.desai@gmail.com' WHERE ID = 3;  **DELETE** FROM student WHERE ID = 3;  DCL– Discuss – Full list of commands with brief description  **GRANT and REVOKE**  TCL– Discuss – Full list of commands with brief description  **COMMIT and ROLLBACK**  Discussion on types Of Constraints (in brief)   * **NOT NULL** - Ensures that a column cannot have a NULL value * **UNIQUE** - Ensures that all values in a column are different * **PRIMARY KEY** - A combination of a NOT NULL and UNIQUE. Uniquely identifies each row in a table * **FOREIGN KEY** - Uniquely identifies a row/record in another table. * **CHECK** - Ensures that all values in a column satisfy a specific condition. | 20 |
| 3(P1) | Database management system | Intro to this course (Ulka and saleel-before recording discuss with sir)  Pre req. – For whom this course imp (eg. Application dev, DBA’s, Data Mining  Career imp - to become DBA, Data Sciten, etc.  Why this course is easy to learn (Section of course, number of sessions, total duration course duration 30hrs.)  Examples been covered. (number of databases, tables discuses)  Note: To be record at the last, content continually updated. | 5 |
| 4(P3) | Introduction to RDBMS | TODO: | 8 |
| 5(P3) | File System vs DBMS | Concepts of Flat Files  Concepts of DBMS  Advantage of File-oriented system  Backup, Data retrieval, Editing, Remoteaccess, Sharing  Disadvantage of File-oriented system  Data redundancy, Datainconsistency, Limiteddatasharing, Datasecurity  Advantage of DBMS  Improveddatasharing, Improveddatasecurity, Minimizeddatainconsistency, Minimized dataredundancy  Disadvantage of DBMS   * Cost of Hardware and Software of a DBMS is quite high * Most database management systems are often complex systems, so the training for users to use the DBMS is required. * All data is integrated into a single database which can be damaged because of electric failure or database is corrupted on the storage device.   Slides with graphics explaining the above concepts (good explanatory images)  Note: Content continually updated. | 5 |
| 6(P1) | Schema in databases | Intro to schema  A **schema** is a collection of database objects including **tables**, **views**, **triggers**, **stored procedures**, **indexes**, etc. A **schema** is associated with a username which is known as the **schema** owner, who is the owner of the logically related database objects.  Defining and creating schema  To create a new database in MySQL, you use the CREATE DATABASE statement with the following syntax:  CREATE {DATABASE | SCHEMA} [IF NOT EXISTS] *db\_name*  e.g.   * CREATE SCHEMA IF NOT EXISTS myDB; * CREATE DATABASE IF NOT EXISTS myDB;   To select a particular database to work with you issue the USE statement with the follows syntax :  USE *db\_name*  The USE statement tells MySQL to use the named database as the default (current) database for subsequent statements. The named database remains the default until the end of the session or another USE statement is issued.  e.g.   * USE myDB;   Managing and moving objects between schemas  Note: Moving objects from one schema to another schema in MySQL is not possible  Note: Slides required, live examples/demos ( Create Schema/database in MySQL is same, Use schema, moving objects (in MS-SQL possible)from one schema to another schema) | 10 |
| 7(P1) | DDL Commands | **DDL** (Data Definition Language) statements or commands are used to define and modify the database structure of your tables or schema.  **CREATE** TABLE statement is used to specify the layout of your table  The table creation command requires the following details.   * Name of the table * Name of the fields * Definitions for each field   syntax to create a MySQL table is as follows  CREATE TABLE table\_name (column\_name column\_type);  CREATE TABLE student (  ID INT(11) PRIMARY KEY,  firstName VARCHAR(45),  lastName VARCHAR(45),  DoB DATE  );  **ALTER** command is very useful when you want to change a name of any table field or if you want to add or delete an existing column in a table.   * ALTER TABLE student ADD COLUMN emailID VARCHAR(128); * ALTER TABLE student CHANGE ID studentID INT; * ALTER TABLE student DROP COLUMN emailID;   DROP command, it is very easy to drop an existing MySQL table, but you need to be very careful while deleting any existing table because the data lost will not be recovered after deleting a table.   * DROP TABLE student;   Disc on CREATE, ALTER, DROP  Disc this commands with ref to tables |  |
| 8(P1) | Candidate and Primary key | What is a candidate key?  **Candidate Key** – A Candidate Key can be any column or a combination of columns that can qualify as unique key in database. There can be multiple Candidate Keys in one table. Each Candidate Key can qualify as Primary Key.  What is a primary key?  **Primary Key**: Primary Key is a set of attributes (or attribute) which uniquely identify the tuples in relation or table. There is one and only one primary key in any relationship.  Here in **student** table you can choose either **ID**, **pan\_card**, or **emailID** columns as primary key column, here **ID** cab be a preferable choice.  Diff between CK and PK   |  |  | | --- | --- | | **Primary Key** | **Candidate Key** | | There can be only one primary key in any relation. | There can be more than one candidate key in a relation. | | Primary key can not contain NULL value. | Candidate key can have NULL value. | | A primary key is a candidate key. | It not compulsory that each candidate key can be a primary key. | | Primary key specifies the important attribute for the relation. | Candidate specifies the key which can qualify for primary key. |   Points to consider to make column PK and CK | 10 |
| 9(P1) | Primary Key | Imp of PK  A primary key is one of the most important steps in good database design. A primary key is a special column (or set of combined columns) in a relational database table, that is used to uniquely identify each record. Each database table needs a primary key.  PRIMARY key must follow some rules.   * A primary key cannot be NULL. * A primary key value must be unique. * A table has only one primary key.   Examples of PK  CREATE TABLE student (  ID INT(11) PRIMARY KEY,  firstName VARCHAR(45),  lastName VARCHAR(45),  DoB DATE,  emailID VARCHAR(145)  );  PK and AUTO\_INCREMENT  CREATE TABLE student (  ID INT(11) PRIMARY KEY AUTO\_INCREMENT,  firstName VARCHAR(45),  lastName VARCHAR(45),  DoB DATE,  emailID VARCHAR(145)  );  PK data typemostly it must be INT(eg ID), (Eg email varchar PK)  Using multiple columns as PK (Composite Key)  **Composite key**, or **composite primary key**, refers to cases where more than one column is used to specify the **primary key** of a table. In such cases, all foreign **keys** will also need to include all the columns in the **composite key**.  Note that the columns that make up a **composite key** can be of different data types.  CREATE TABLE student (  ID INT(11),  firstName VARCHAR(45),  lastName VARCHAR(45),  DoB DATE,  emailID VARCHAR(145),  PRIMARY KEY (ID, emailID)  );  Note: With Slides and Examples (student Table R= {ID, firstName, lastName, DoB, emailID }) | 15 |
| 10(P1) | Foreign Key | Concepts of FK  A **FOREIGN KEY** is a **key** used to link two tables together. A **FOREIGN KEY** is a field (or collection of fields) in one table that refers to the PRIMARY **KEY** in another table. The table containing the **foreign key** is called the child table, and the table containing the candidate **key** is called the referenced or parent table.   * The table containing the FOREIGN KEY is referred to as the child table, and the table containing the PRIMARY KEY (referenced key) is the parent table. * PARENT and CHILD tables must use the same storage engine, * and they cannot be defined as temporary tables.   Examples of FK  CREATE TABLE student (  ID INT(11) PRIMARY KEY,  firstName VARCHAR(45),  lastName VARCHAR(45),  DoB DATE,  emailID VARCHAR(145)  );  CREATE TABLE student\_address (  studentID INT PRIMARY KEY,  address VARCHAR(45),  city VARCHAR(45) ,  FOREIGN KEY (studentID) REFERENCES student(ID)  );  CREATE TABLE student (  ID INT(11) PRIMARY KEY,  firstName VARCHAR(45),  lastName VARCHAR(45),  DoB DATE,  emailID VARCHAR(145)  );  CREATE TABLE student\_hobbies (  ID INT PRIMARY KEY,  studentID INT,  name VARCHAR(45),  FOREIGN KEY (studentID) REFERENCES student(ID)  );  Data type of PK/FK must be same  FOREIGN key condition must follow some rules.   * A foreign key column can have a different name from its primary key column. * DataType of primary key and foreign key column must be same. * It ensures rows in one table have corresponding rows in another. * Foreign keys can be NULL value even though primary keys can not.   Note: With Slides and Examples (student Table R= {ID(PK), firstName, lastName, DoB, emailID } , student\_address Table, student\_hobbies table}) – insert 4-5 demo records  Structure view and Browser view  URL: <https://dev.mysql.com/doc/workbench/en/wb-creating-eer-diagram.html> |  |
| 11(P1) | Introduction to DML commands | **DML** stands for Data Manipulation Language. It is a language used for **inserting**, **selecting**,**updating** and **deleting** data in a tables.  Use of INSERT, SELECT, UPDATE, DELETE and TRUNCATE withsingle table (student table)  The **INSERT** **INTO** s**tatement** is used to add new rows of data to a table in the database.   * INSERT INTO student VALUES (5, 'sohan', 'kumar', '1999-09-19', ‘sohankumar@gmail.com'); * INSERT INTO student\_address VALUES (5, 'station road', 'baroda'); * INSERT INTO student\_hobbies VALUES (7, 1, 'football');   The **SELECT statement** is used to fetch the data from a database table which returns this data in the form of a result table.   * SELECT \* FROM student; * SELECT \* FROM student\_address; * SELECT \* FROM student\_hobbies;   The **UPDATE statement** is used to **update** the data of an existing table in database. We can **update** single columns as well as multiple columns using **UPDATE statement** as per our requirement.   * UPDATE student SET email =NULL; * UPDATE student SET emailID = ' ramesh.patel123@yahoomail.com' where ID = 1;   The **DELETE Statement** is used to delete existing records from a table. We can delete a single record or multiple records depending on the condition.   * DELETE FROM student; * DELETE FROM student where ID = 1;   The TRUNCATE TABLE command deletes the data inside a table, but not the table itself.   * TRUNCATE TABLE student;   Use of above commands with related tables (PK/FK table – student - student\_address)   * INSERT INTO student\_address VALUES (1, 'paud road', 'pune'); * INSERT INTO student\_address VALUES (2, 'M.G. road', 'baroda'); * INSERT INTO student\_address VALUES (3, 'k.k road', 'surat'); * INSERT INTO student\_address VALUES (104, 'station road', 'baroda');   Referential integrity (Directly adding a record in student\_address, deleting a student record,  Similar for student\_hobbies table)   * INSERT INTO student\_hobbies VALUES (1, 1, 'running'); * INSERT INTO student\_hobbies VALUES (2, 1, 'reading'); * INSERT INTO student\_hobbies VALUES (3, 2, 'football'); * INSERT INTO student\_hobbies VALUES (4, 2, 'running'); * INSERT INTO student\_hobbies VALUES (105, 3, 'watching movies'); * INSERT INTO student\_hobbies VALUES (104, 4, 'gaming');   Note: With Slides and Examples (student Table R= {ID(PK), firstName, lastName, DoB, emailID } , student\_address Table, student\_hobbies table})  Structure view and Brows view | 20 |
| 12(P2) | Working with ER models | What are ER models  Working with MySQL ER model Tool.  Benefits of documentation  Third party ER database documentations tools (toad from DELL, NaviCat)  Note: With Slides and Examples | 5-10 |
| 13(P3) | Using ER model for Project Documentations | Attributes of ER models  TODO |  |
| 14(P1) | Introduction to Relationships | Basic intro to all relationships  Note: With Slides with relationship definition and ER images | 5 |
| 15(P1) | One-to-one Relationship | Discussion of one-to-one relationships  A *one-to-one* relationship between two tables means that a row in one table can only relate to one row in the table on the other side of their relationship and vice versa. This is the least common database relationship.  Design intents for creating one-to-one relationships (situations like student\_address, one more example to be added)  A one-to-one relationship is a type of cardinality that refers to the relationship between two entities A and B in which one element of A may only be linked to one element of B, and vice versa.  CREATE TABLE student (  ID INT(11) PRIMARY KEY,  firstName VARCHAR(45),  lastName VARCHAR(45),  DoB DATE,  emailID VARCHAR(145)  );  CREATE TABLE student\_address (  ID int PRIMARY KEY,  studentID INT UNIQUE NOT NULL,  address VARCHAR(45),  city VARCHAR(45) ,  FOREIGN KEY (studentID) REFERENCES student(ID)  );  CREATE TABLE person  (  ID INT PRIMARY KEY ,  name VARCHAR(45),  emailID VARCHAR(128)  );  CREATE TABLE passportDetails  (  passportID INT PRIMARY KEY,  passport\_Number VARCHAR(255),  person\_ID INT UNIQUE,  FOREIGN KEY (person\_ID) REFERENCES person(ID)  );  How to create one-to-one relationship (T1-PK/T2-PK&FK)      Note: With Slides and Examples | 10 |
| 16(P1) | One-to-many Relationship | Discussion of one-to-many relationships  A *one-to-many* relationship between two tables means that a row in one table can have one or more row in the table on the other side of their relationship.  A *one-to-many* relationship is a type of cardinality that refers to the relationship between two entities A and B in which an element of A may be linked to many elements of B, but a member of B is linked to only one element of A.  Design intents for creating one-to-many relationships (situations like student\_hobbies, one more example of **invoice** and **invoice\_items**) (Every one invoice must have at least one item))  CREATE TABLE student (  ID INT(11) PRIMARY KEY,  firstName VARCHAR(45),  lastName VARCHAR(45),  DoB DATE,  emailID VARCHAR(145)  );  CREATE TABLE student\_hobbies (  ID INT PRIMARY KEY,  studentID INT,  name VARCHAR(45),  FOREIGN KEY (studentID) REFERENCES student(ID)  );    How to create one-to-many relationship  Note: With Slides and Examples | 10 |
| 17(P1) | Many-to-many Relationship | Discussion of many-to-many relationships  A many-to-many relationship is a type of cardinality that refers to the relationship between two entities A and B in which A may contain a parent instance for which there are many children in B and vice versa.  Design intents for creating many-to-many relationships (situations like student, course, and course\_student)  How to create many-to-many relationship  Note: With Slides and Examples | 10 |
| 18(P4) | Many-to-many Relationship - 2 | Solving advance problem statements | TODO |
| 19(P2) | Introduction to Normalization | Concepts of Normalization  Reasons to normalize the database.  Brief on all normalization levels  Note: With Slides-good graphics | 10 |
| 20(P2) | Normalization | Insertion, Updating, and Deletion Anomaly  Note: With Slides and example | 5-10 |
| 21(P2) | First form of Normalization | Def of first form of normalization (1NF)  Understanding of closure of functional dependency  Understanding functional dependency and its properties.  Note: With Slides-good graphics and example | 10 |
| 22(P2) | Functional Dependency | Disc on Function Dependency  Closure of FD  Properties of FD  Note: With Slides-good graphics and example |  |
| 23(P2) | Second Normalization | Dis on 2NF  Note: With Slides-good graphics and example |  |
| 24(P2) | Third Normalization Form | Dis on 3NF  Problem statement for 3NF. (on Faculty/Room/Timing)  Note: With Slides-good graphics and example |  |
| 25(P4) | Boyce Codd Normalization | Dis on BCNF |  |
| 26(P4) | Fifth Normalization Form | Dis on 5thNF  Loss less and loose decomposition  Minimal coverage  Note: With Slides-good graphics and example |  |
| 27(P2) | FAQ on Normalization | Recap on Normalization (1,2 and 3NF)  15 Interview Que and Ans on Normalization.  Note: With Slides-good graphics and example |  |
| 28(P1) | Introduction to Joins | Dis on JOINS  The **SQL Joins** clause is **used** to combine records from two or more tables in a database. A **JOIN** is a means for combining fields from two tables by using values common to each.  Need for Joins  Types of Joins   * Cartesian or Product Join – Cross Join * Equijoin – Inner Join * Natural Join * Outer Join – Right Outer Join, Left Outer Join * Self-Join   Cartesian or Product joins are joins without a join condition. Each row of one table is combined with each row of another table. The result is referred to as a Cartesian product.  EQUI JOIN performs a JOIN against equality or matching column(s) values of the associated tables. An equal sign (=) is used as comparison operator in the where clause to refer equality.  The **LEFT** **JOIN** keyword returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is NULL in the right side when there is no match.  The **RIGHT** **JOIN** keyword returns all rows from the right table (table2), with the matching rows in the left table (table1). The result is NULL in the left side when there is no match.  Note: With Slides-good graphics and example | 10 |
| 29(P1) | Inner/ equi Joins | Dis on Inner/equi Joins  The inner join is one of the most commonly used joins in SQL. The inner join clause allows you to query data from two or more related tables.  **INNER** **JOIN** performs a JOIN against equality or matching column(s) values of the associated tables. An equal sign (=) is used as comparison operator in the **ON** clause to refer equality.  **EQUI** **JOIN** performs a JOIN against equality or matching column(s) values of the associated tables. An equal sign (=) is used as comparison operator in the **WHERE** clause to refer equality.  Dis on syntax  SELECT column-list from <table\_references>, <table\_references> WHERE table1.column-name = table2.column-name  SELECT column-list from <table\_references> [INNER] JOIN <table\_references> ON table1.column-name = table2.column-name  Situations to use Inner Joins.  **Use** an  **INNER JOIN** when you need to match rows from two tables. Rows that match remain in the result, those that don't are rejected. The match condition is commonly called the **join** condition. Inner/ equi Join returns rows when there is at least one match in both tables.  Select appropriate tables (Student, student\_address)  Diff between Inner/equi and Natural join   |  |  | | --- | --- | | **INNER JOIN** | **NATURAL JOIN** | | Inner Join joins two table on the basis of the column which is explicitly specified in the ON clause. | Natural Join joins two tables based on same attribute name. | | In Inner Join, The resulting table will contain all the attribute of both the tables including duplicate columns also | In Natural Join, The resulting table will contain all the attributes of both the tables but keep only one copy of each common column | | In Inner Join, only those records will return which exists in both the tables | Same as Inner Join | | SYNTAX: SELECT \* FROM table1 INNER JOIN table2 ON table1.Column\_Name = table2.Column\_Name; | SYNTAX: SELECT \* FROM table1 NATURAL JOIN table2; |   Note: With Slides-good graphics and example | 10 |
| 30(P1) | Natural Join | Dis on Natural Join (Create new tables for Natural Join) keep the tables ready.  The **NATURAL** **JOIN** is such a join that performs the same task as an **INNER** **JOIN**. **NATURAL** **JOIN** does not use any comparison operator. We can perform a **NATURAL** **JOIN** only if there is at least one common attribute that exists between two relations. In addition, the attributes must have the same name and domain. When this join condition gets applied always the duplicates of the common columns get eliminated from the result.  Note:   * If the column-names are not same, then NATURAL JOIN will work as CROSS JOIN. * A NATURAL JOIN can be used with a LEFT OUTER join, or a RIGHT OUTER join.   CREATE TABLE food (  item\_id INT,  item\_name VARCHAR(45),  item\_unit VARCHAR(45),  company\_id INT );  CREATE TABLE company (  company\_id INT,  company\_name VARCHAR(45),  company\_city VARCHAR(45)  );  Diff between Natural join and Inner/equi Join.   |  |  | | --- | --- | | **NATURAL JOIN** | **INNER JOIN** | | Natural Join joins two tables based on same attribute name. | Inner Join joins two table on the basis of the column which is explicitly specified in the ON clause. | | In Natural Join, The resulting table will contain all the attributes of both the tables but keep only one copy of each common column | In Inner Join, The resulting table will contain all the attribute of both the tables including duplicate columns also | | In Inner Join, only those records will return which exists in both the tables | In Inner Join, only those records will return which exists in both the tables | | SYNTAX: SELECT \* FROM table1 NATURAL JOIN table2; | SYNTAX: SELECT \* FROM table1 INNER JOIN table2 ON table1.Column\_Name = table2.Column\_Name; |   Note: With Slides-good graphics and example | 5 |
| 31(P1) | Self Join | Dis on Self Join  A **self join** is a **join** in which a table is joined with itself (which is also called Unary relationships), especially when the table has a FOREIGN KEY which references its own PRIMARY KEY. To **join** a table itself means that each row of the table is combined with itself and with every other row of the table.  In this example, we referenced to the  staffs table twice: one as e for the employees and the other as m for the managers. The join predicate matches employee and manager relationship using the values in the e.empno and m.mgr columns.   * SELECT e.ename EmployeeName, m.ename ManagerName FROM emp m, emp e WHERE m.empno = e.mgr;   Note: With Slides-good graphics and example |  |
| 32(P1) | Left Outer Join | Dis on Left Outer Join  The LEFT JOIN clause allows you to query data from multiple tables. It returns all rows from the left table and the matching rows from the right table. If no matching rows found in the right table, NULL are used.  Suppose, we want to join two tables: A and B. SQL left outer join returns all rows in the left table (A) and all the matching rows found in the right table (B). It means the result of the SQL left join always contains the rows in the left table. . If no matching rows found in the right table, NULL are displayed.  SELECT column-list from <table\_references> **LEFT [OUTER ] JOIN** <table\_references> **ON** table1.column-name = table2.column-name  Examples Student / CourseFees / Library  CREATE TABLE student (  ID INT(11) PRIMARY KEY,  firstName VARCHAR(45),  lastName VARCHAR(45),  DoB DATE,  emailID VARCHAR(145)  );  CREATE TABLE CourseFees (  ID INT(11) PRIMARY KEY AUTO\_INCREMENT,  studentID INT,  fees INT  );  INSERT INTO student VALUES (1, 'ramesh', 'patel', '1999-10-17', 'ramesh.patel@gmail.com');  INSERT INTO student VALUES (2, 'rajesh', 'mehta', '2000-12-20', 'rajesh.mehta@gmail.com');  INSERT INTO student VALUES (3, 'vipul', 'shah', '2001-07-19', 'shahvipul@yahoomail.com');  INSERT INTO student VALUES (4, 'kamlesh', 'kaka', '2002-11-26', 'kamlesh.kaka@gmail.com');  INSERT INTO student VALUES (5, 'raj', 'patel', '1999-10-17', 'raj123@gmail.com');  INSERT INTO student VALUES (6, 'sam', '', '2002-11-12', 'sam665@gmail.com');  INSERT INTO student VALUES (7, 'vimal', 'kumar', '2003-01-29', 'vimalkumar@yahoomail.com');  INSERT INTO student VALUES (8, 'suraj', 'nath', '2002-11-26', 'nathsuraj@gmail.com');  INSERT INTO coursefees VALUES (default, 2, 20000);  INSERT INTO coursefees VALUES (default, 3, 25000);  INSERT INTO coursefees VALUES (default, 4, 12000);  INSERT INTO coursefees VALUES (default, 5, 15000);  Note: With Slides-good graphics and example |  |
| 33(P1) | Right Outer Join | Dis on Right Outer Join  The RIGHT JOIN combines data from two or more tables. The RIGHT JOIN clause starts selecting data from the right table and matching with the rows from the left table. The RIGHT JOIN returns a result set that includes all rows in the right table, whether or not they have matching rows from the left table. If a row in the right table does not have any matching rows from the left table, the column of the left table in the result set will have nulls.  Suppose, we want to join two tables: A and B. Right outer join returns all rows in the right table (A) and all the matching rows found in the left table (B). It means the result of the SQL right join always contains the rows in the right table. . If no matching rows found in the left table, NULL are displayed.  The following example the RIGHT OUTER JOIN of two tables T1(AC-1, AC-2, AC-3, AC-4, AC-5) and T2(C-1, C-2, C-3, C-4). The RIGHT JOIN will match rows from the T1 table with the rows from T2 table using patterns:   * SELECT student.ID, firstName, lastName, title, issueDate FROM student RIGHT OUTER JOIN library ON student.ID = library.studentID;   Examples Student/CourseFees/Library  Note: With Slides-good graphics and example |  |
| 34(P1) | Cross Join / Cartesian Join | Dic on Cross Join and Cartesian 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.  The CROSS JOIN gets a row from the first table (T1) and then creates a new row for every row in the second table (T2). It then does the same for the next row for in the first table (T1) and so on.  E.g. multi location chain of shops and their product inventory.  URL: <https://www.sqlservertutorial.net/sql-server-basics/sql-server-cross-join/>  Note: With Slides-good graphics and example |  |
| 35(P1) | UNION | Definition of UNION and UNION ALL  The **UNION** operator is used to combine the result sets of 2 or more SELECT statements. It removes duplicate rows between the various SELECT statements. Each SELECT statement within the **UNION** operator must have the same number of fields in the result sets with similar data types.  The **Union ALL** operator combines the results of two or more queries into a single result set that includes all the rows that belong to all queries in the Union. In simple terms, it combines the two or more row sets and keeps duplicates.  E.g. of union with diagram  URL: <https://www.sqlshack.com/sql-union-overview-usage-and-examples/#:~:text=The%20Union%20operator%20combines%20the,has%203%2C4%2C5.>  Note: With Slides-good graphics and example |  |
| 36(P1) | Difference between Delete and Truncate | What DELETE does  **1. DELETE :** DELETE is a DML(Data Manipulation Language) command and is used when we specify the row(tuple) that we want to remove or delete from the table or relation. The DELETE command can contain a WHERE clause. If **WHERE** clause is used with DELETE command then it remove or delete only those rows(tuple) that satisfy the condition otherwise by default it removes all the tuples(rows) from the table.  What truncate does  **2. TRUNCATE :** TRUNCATE is a DDL(Data Definition Language) command and is used to delete all the rows or tuples from a table. Unlike the DELETE command, TRUNCATE command does not contain a WHERE clause. In the TRUNCATE command, the transaction log for each deleted data page is recorded. Unlike the DELETE command, the TRUNCATE command is fast and we can’t rollback the data after using the TRUNCATE command.   |  |  |  | | --- | --- | --- | | S.NO | DELETE | TRUNCATE | | 1. | The DELETE command is used to delete specified rows(one or more). | While this command is used to delete all the rows from a table. | | 2. | It is a DML(Data Manipulation Language) command. | While it is a DDL(Data Definition Language) command. | | 3. | There may be WHERE clause in DELETE command in order to filter the records. | While there may not be WHERE clause in TRUNCATE command. | | 4. | In the DELETE command, a tuple is locked before removing it. | While in this command, data page is locked before removing the table data. | | 5. | We can rollback the data even after using DELETE command. | While in this command, we can’t rollback. | | 6. | DELETE command is slower than TRUNCATE command. | While TRUNCATE command is faster than DELETE command. |   When to use delete and truncate with e.g. and demo.  Note: With Slides-good graphics and example | 5 |
| 37(P1) | Queries and Sub-queries | Dis on Queries (SELECT)  Disc on Sub-queries (What is a sub-query)  Why to use sub-queries.  E.g. of Using of sub-query  Note: With Slides-good graphics and example |  |
| 38(P1) | Queries and Nested Queries | Dis on nested Queries (SELECT)     * A Subquery or Inner query or a Nested query is a query within another SQL query and embedded within the WHERE clause. * A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved. * Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.   Disc on nested queries (What is a nested query)  Why to use nested queries.  **Important Rule:**   * A subquery can be placed in a number of SQL clauses like WHERE clause, FROM clause, HAVING clause. * You can use Subquery with SELECT, UPDATE, INSERT, DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc. * A subquery is a query within another query. The outer query is known as the main query, and the inner query is known as a subquery. * Subqueries are on the right side of the comparison operator. * A subquery is enclosed in parentheses. * In the Subquery, ORDER BY command cannot be used. But GROUP BY command can be used to perform the same function as ORDER BY command.   A subquery, also known as a nested query, subquery is a SELECT query embedded within the WHERE or HAVING clause of another SQL query. The data returned by the subquery is used by the outer statement in the same way a literal value would be used.  **Important Rule:**   * A subquery can be placed in a number of SQL clauses like WHERE clause, FROM clause, HAVING clause. * You can use Subquery with SELECT, UPDATE, INSERT, DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc. * A subquery is a query within another query. The outer query is known as the main query, and the inner query is known as a subquery. * Subqueries are on the right side of the comparison operator. * A subquery is enclosed in parentheses.   Types of subquery   * The Subquery as Scalar Operand – SELECT clause * Comparisons using Subqueries – WHERE / HAVING clause (Single row subquery) * Subqueries in the FROM Clause – Inline Views   E.g. of Using of nested query  Note: With Slides-good graphics and example |  |
| 39(P1) | Queries and sub-queries using Group by clause | Dis on Group by (SELECT)  Disc on Group by)  Why to use group by clause.  E.g. of Using of group by  Note: With Slides-good graphics and example |  |
| 40(P1) | Having | Dis on Having (SELECT)  Disc on Having )  Why to use having clause.  E.g. of Using of having clause  Note: With Slides-good graphics and example |  |
| 41(P1) | Queries and sub-queries using in and not in clause | Dis on in and not in clause (SELECT)  Disc on in and not in clause)  Why to use in and not in clause.  E.g. of Using of in and not in clause  Note: With Slides-good graphics and example |  |
| 42(P1) | Queries and sub-queries using exists and not exists | Dis on exists and not exists clause (SELECT)  Disc on exists and not exists clause)  Why to use exists and not exists clause.  E.g. of Using of exists and not exists in clause  Note: With Slides-good graphics and example |  |
| 43(P1) | Aggregate Functions | Aggregate functions retrieve a single **value** after performing a calculation on a set of values. In general, aggregate functions ignore null values. Often, aggregate functions are use with GROUP BY clause of the SELECT statement.  Dis on SUM, AVG, COUNT , MIN, MAX  **SUM function**  The SUM function returns the sum of all the values in the specified column. SUM works on numeric fields only. Null values are excluded from the result returned.   * SELECT sum(invoiceAmount) "Total invoice amount for all customer " FROM invoice; * SELECT customerID, sum(invoiceAmount) "Total invoice amount for each customer " from invoice group by customerID;   **AVG function**  The AVG function returns the average of the values in a specified column. Just like the SUM function, it works only on numeric data types.   * SELECT avg(invoiceAmount) "Average invoice amount for all customer" FROM invoice; * SELECT customerID, avg(invoiceAmount) "Average invoice amount for each customer" from invoice group by customerID;   **COUNT Function**  The COUNT function returns the total number of values in the specified field. It works on both numeric and non-numeric data types. All aggregate functions by default exclude nulls values before working on the data.  COUNT (\*) is a special implementation of the COUNT function that returns the count of all the rows in a specified table. COUNT (\*) also considers Nulls and duplicates.   * SELECT count(ID) "Total invoices" FROM invoice; * SELECT customerID, count(ID) "Total invoices for each customer" FROM invoice GROUP BY customerID;   **MIN function**  The MIN function returns the smallest value in the specified table field.   * SELECT min(invoiceAmount) "Minimum invoices amount" FROM invoice; * SELECT customerID, min(invoiceAmount) "Minimum invoices for each customer" FROM invoice group by customerID;   **MAX function**  Just as the name suggests, the MAX function is the opposite of the MIN function. It returns the largest value from the specified table field.   * SELECT max(invoiceAmount) "Maximum invoices amount" FROM invoice; * SELECT customerID, max(invoiceAmount) " Maximum invoices for each customer" FROM invoice group by customerID;   2-3 e.g. of each  Live DEMO;  Note: With Slides-good graphics and example |  |
| 44(P1) | Introduction to PL/SQL | Dis on PL/SQL  PL/SQL stands for “Procedural Language extensions to the Structured Query Language”. SQL is a popular language for both querying and updating data in the relational database management systems (RDBMS). PL/SQL is a block structured language that enables developers to combine the power of SQL with procedural statements.  **Features of PL/SQL:**   * PL/SQL is basically a procedural language, which provides the functionality of decision making, and iteration. * PL/SQL can execute a number of queries in one block using single command. * PL/SQL unit such as procedures, functions, packages, triggers, and types, which are stored in the database for reuse by applications. * PL/SQL provides a feature to handle the exception which occurs in PL/SQL block known as exception handling block.   Note: With Slides-good graphics and example |  |
| 45(P1) | Using stored procedure | Need for SP  A stored procedure is a subroutine available to applications that access a relational database management system. Such procedures are stored in the database data dictionary. 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.  How to create Stored Procedure  DELIMITER //  **CREATE** **PROCEDURE** SP\_NAME()  **BEGIN**  **# statements**;  **END** //  DELIMITER ;  Namening conventions of SP.  IN, OUT, and INOUT parameters  In MySQL, a parameter has one of three modes: IN,OUT, or INOUT.  **IN parameters**  IN is the default mode. When you define an IN parameter in a stored procedure, the calling program has to pass an argument to the stored procedure.  **OUT parameters**  The value of an OUT parameter can be changed inside the stored procedure and its new value is passed back to the calling program.  **INOUT parameters**  An INOUT parameter is a combination of IN and OUT parameters. It means that the calling program may pass the argument, and the stored procedure can modify the INOUT parameter, and pass the new value back to the calling program.   * Write a stored procedure to get the student ID from the user, print student data along with his qualification details.   DROP PROCEDURE if EXISTS getStudentByID;  delimiter $$  CREATE PROCEDURE getStudentByID(in para1 INT)  begin  SELECT \* FROM student, student\_qualifications WHERE student.ID = student\_qualifications.studentID and student.ID = para1;  end $$  delimiter ;   * Write a stored procedure to get the student ID from the user, return his namefirst, namelast, and his address .   DROP PROCEDURE if EXISTS getStudentWithAddress;  delimiter $$  CREATE PROCEDURE getStudentWithAddress(in p\_ID int, out p\_namefirst varchar(45), out p\_namelast varchar(45), out p\_address varchar(128))  begin  SELECT namefirst, namelast, address INTO p\_namefirst, p\_namelast, p\_address FROM student, student\_address WHERE student.ID= student\_address.studentID and student.ID=p\_ID;  end $$  delimiter ;  Note: With Slides-good graphics and example |  |
| 46(P1) | Using Functions | Need for Functions  A function is a subroutine available to applications that access a relational database management system. Such functions are stored in the database data dictionary. A function is a block of organized, reusable code that is used to perform a single action.  How to create Functions  DELIMITER //  **CREATE** **FUNCTION** function\_name(func\_parameter1, func\_parameter2, ..) RETURNS datatype  **BEGIN**  **# statements**;  **END** //  DELIMITER ;   * First, you specify the name of the stored function after CREATE FUNCTION clause. * Second, you list all parameters of the stored function inside the parentheses. * Third, the RETURNS clause may be specified only for a *FUNCTION*, for which it is mandatory. It indicates the return type of the function, and the function body must contain a RETURN value statement. * Write a stored function that will generate studentID.   DROP FUNCTION IF EXISTS generateStudentID;  delimiter $$  CREATE FUNCTION generateStudentID() RETURNS INT  begin  declare NextNumber INT;  SELECT max(ID) + 1 INTO NextNumber FROM student;  RETURN(NextNumber);  end $$  delimiter ;   * Write a stored function to get the student ID from the user and return on which day he was born.   DROP FUNCTION IF EXISTS getDayNameOfStudent;  delimiter $$  CREATE FUNCTION getDayNameOfStudent(SID int) RETURNS VARCHAR(20)  begin  DECLARE v\_dayname VARCHAR(20);  SELECT dayname(dob) INTO v\_dayname FROM student WHERE ID = SID;  RETURN(v\_dayname);  end $$  delimiter ;  Namening conventions of Functions.  Note: With Slides-good graphics and example |  |
| 47(P1) | Using Triggers | Need for Triggers  How to create Triggers  Namening conventions of Triggers.  Note: With Slides-good graphics and example |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |