**SQL Practice Sheet**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1. Data Query Language**  Command**: SELECT**  **SELECT** expressions **FROM** TABLE.NAME **where** conditions;  where clause: Used for filtering based on some conditions  E.g.,   * Selecting all columns: Select \* FROM Customers; * Selecting specific columns: Select CustomerName, City FROM Customers; * Selecting distinct values in a column: Select **DISTINCT** Country FROM Customers; * Counting distinct values in a column: Select **COUNT**(DISTINCT Country) FROM Customers; * Filtering Customers based on a specific Country: Select \* FROM Customers **where** Country = ‘Mexico’   **Operators Used in where clause**   |  |  |  | | --- | --- | --- | | No. | Operator | Description | | 1. | =, <, <= ,> , >= | Equals, less than, less than or equal, Greater than, Greater than or equal | | 2. | <>, != | Not Equals | | 3. | BETWEEN, LIKE, IN | BETWEEN: Between a certain range  LIKE: Search for a pattern  IN: To specify multiple possible values for a column | | 4. | AND, OR, NOT | Same as logical. | | 5. | ALL, ANY | ALL: It compares a value to all values in a Value Set.  ANY: It compares the values in the list according to the condition. | | 6. | EXISTS | It is used to search for the presence of a row in a specified table. |  * Select \* FROM Products WHERE Price **BETWEEN** 50 AND 60; * Filtering Products based on City that contains ‘st’: Select \* FROM Products WHERE City **LIKE** ‘%st%’; * Filtering Products based on City that starts with ‘st’: Select \* FROM Products WHERE City **LIKE** ‘st%’; * Select \* FROM Customers WHERE City **IN** (‘Paris’, ‘London’); * Select \* FROM Customers WHERE City = ‘Berlin’ **AND** CustomerName LIKE ‘G%’ * Select \* FROM Customers WHERE City = ‘Berlin’ **OR** CustomerName LIKE ‘G%’ * Select \* FROM Customers WHERE **NOT** Country = ‘Spain’; * Select \* FROM Customers WHERE CustomerName **NOT LIKE** ‘A%’ * Select \* FROM Customers WHERE CustomerID **NOT BETWEEN** 10 AND 60; * Select \* FROM Customers WHERE City **NOT IN** (‘Paris’, ‘London’); * Select \* FROM Customers WHERE **NOT** CustomerID > 50; * **Select** SupplierName **FROM** Suppliers **WHERE** **EXISTS** (**Select** ProductName **FROM** Products **WHERE** Products.SupplierID = Suppliers.supplierID AND Price = 22); * Select ProductName **FROM** Products **WHERE** ProductID = **ANY** (Select ProductID FROM OrderDetails WHERE Quality > 99); * Select ProductName **FROM** Products **WHERE** ProductID = **ALL** (Select ProductID FROM OrderDetails WHERE Quality > 99);   **ORDER BY keyword:** Used to sort the result set in ascending or descending order;   * Ascending by default: Select \* FROM Products ORDER BY Price; * Descending: Select \* FROM Products ORDER BY Price DESC; * Order By (several column): Select \* FROM Customers ORDER BY Country, CustomerName; * Using Both ASC & DESC: Select \* FROM Customers ORDER BY Country ASC, CustomerName DESC;   **Note:**   1. Not all database system supports the **SELECT TOP** clause. MySQL supports the **LIMIT** clause to select a limited number of records, while Oracle uses **FETCH FIRST n ROWS ONLY** and **ROWNUM**. 2. We can add a WHERE clause, ORDER BY keyword along with SELECT TOP clause.   **SELECT TOP, LIMIT, FETCH FIRST .. ROWS ONLY clause**   1. *SQL Server:* Select **TOP number | percent** columnName **FROM** table\_name **WHERE** condition; 2. *MySQL:* Select columnName **FROM** tableName **WHERE** condition **LIMIT** number; 3. *Oracle 12:* Select columnName **FROM** tableName **ORDER BY** columnName **FETCH FIRST number ROWS ONLY**; 4. *Older Oracle:* Select columnName **FROM** tableName **WHERE** **ROWNUM** <= number; 5. *Older Oracle:* Select \* FROM (Select columnName **FROM** tableName **ORDER BY** columnName) where **ROWNUM** <= number;   **Examples:**   * Select TOP 3 \* FROM Customers; * Select TOP 50 PERCENT \* FROM Customers; * Select \* FROM Customers LIMIT 3; * Select \* FROM Customers WHERE Country = ‘Germany’ ORDER BY CustomerName DESC LIMIT 3; * Select \* FROM Customers FETCH FIRST 3 ROWS ONLY; * Select \* FROM Customers FETCH FIRST 50 PERCENT ROWS ONLY; * Select \* FROM Customers WHERE Country = ‘Germany’ ORDER BY CustomerName DESC FETCH FIRST 3 ROWS ONLY; |

|  |
| --- |
| **2. Data Manipulation Language (DML)**   * DML Commands are used to modify the database. It is responsible for all form of changes in the database. * The command of DML is not auto – committed i.e., it can’t permanently save all the changes in the database. They can be rollback.   Command**: INSERT**  INSERT INTO tableName (column1, column2...) VALUES (value1, value2…)  INSERT INTO tableName VALUES (val1, val2…)  **Example:**   * INSERT INTO Customers (CustomerName, City, PostalCode) VALUES (‘Shivam’, ‘Delhi’, ‘4006’), (‘Raj’, ‘Pune’, ‘4007’) * INSERT INTO Customers VALUES (‘Shivam’, ‘Delhi’, ‘4006’), (‘Raj’, ‘Pune’, ‘4007’)   Command: **UPDATE**  UPDATE tableName set col1 = val1, col2 = val2… where condition;  **Example:**   * UPDATE Customers **SET** ContactName = ‘sam’, City = ‘Brooklynn’ **WHERE** CustomerID = 1;   Command: DELETE  DELETE FROM tableName WHERE condition;  **Example:**   * DELETE FROM Customers WHERE CustomerName = ‘Sam’; * DELETE FROM Customers; |

**SQL Null Values**

* Use IS NULL and IS NOT NULL operators to test instead of = , <, or >
* Select columnNames FROM tableName **WHERE** columnName **IS NULL;**
* Select columnNames FROM tableName WHERE columnName **IS NOT NULL;**

**Examples**

* Select CustomerName, Contact, Address FROM Customers WHERE Address IS NULL;
* Select CustomerName, Contact, Address FROM Customers WHERE Address IS NOT NULL;

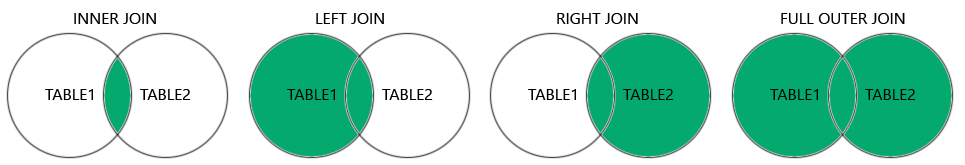
|  |
| --- |
| **SQL Aggregate Functions**   * An aggregate function is a function that performs a calculation on a set of values, & return single value. * Aggregate Functions are often used with GROUP BY clause of the SELECT statement.   **GROUP BY clause**   * GROUP BY clause splits the result set into groups of values & the aggregate function can be used to return single value from each group. * GROUP BY clause follows the WHERE clause in a SELECT statement & precedes ORDER BY clause. * **SELECT** columns **FROM** tableName **WHERE** conditions **GROUP BY** column **ORDER BY** column.   **Example:**  **Select** COUNT (CustomerID), Country **FROM** Customers **GROUP BY** Country **ORDER BY** COUNT(CustomerID) DESC;  **Select** Shippers.ShipperName, COUNT (Orders.OrderID) AS NumberOfOrders **FROM** Orders **LEFT JOIN** Shippers **ON** Orders.ShipperID = Shippers.ShipperID **GROUP BY** ShipperName;  **HAVING clause**   * HAVING clause is used to specify a search condition for a group or an aggregate. * It is used with GROUP BY clause. If we’re not using GROUP BY clause then we can use HAVING clause like a WHERE clause. * **SELECT** column **FROM** table **WHERE** condition **GROUP BY** column **HAVING** condition **ORDER BY** column;   Example:   * **SELECT** COUNT(CustomerID), Country **FROM** Customers **GROUP BY** Country **HAVING** COUNT(CustomerID) > 5; * **SELECT** COUNT(CustomerID), Country **FROM** Customers **GROUP BY** Country **HAVING** COUNT(CustomerID) > 5 ORDER BY COUNT(CustomerID) DESC; * **SELECT** Employees.LastName, COUNT (Orders.OrderID) AS NumberOfOrders **FROM** (Orders INNER JOIN Employees **ON** Orders.EmployeeID = Employees.EmployeeID) **GROUP BY** LastName **HAVING** COUNT (Orders.OrderID) > 10; |

|  |  |  |
| --- | --- | --- |
| **No.** | **Aggregate Functions** | **Description** |
| 1. | MIN () | Returns the smallest value within the selected column.  **MIN** () or **MIN** ([ALL|DISTINCT] expression)  e.g., Select MIN(Price) from Products; |
| 2. | MAX () | Returns the largest value within the selected column.  **MAX** () or **MAX** ([ALL|DISTINCT] expression)  e.g., Select MAX(Price) from Products; |
| 3. | COUNT () | Returns the number of rows in a set.  **COUNT** (\*) or **COUNT** ([ALL|DISTINCT] expression)  **e.g.,**  Select **COUNT** (\*) from Products;  Select **COUNT** (ProductID) from Products where Price > 20;  Select **COUNT** (DISTINCT Price) from Products;  Select **COUNT** (\*) AS [RecordsCount], CategoryID from Products **GROUP BY** CategoryID;  \* - Considers Null values  ColumnName – doesn’t consider null values. |
| 4. | SUM () | Returns the total sum of a numerical column.  **SUM** () or **SUM** ([ALL|DISTINCT] expression)  **e.g.,**  Select **SUM** (Quantity) from OrderDetails;  Select **SUM** (Quantity \* 10) from OrderDetails;  Select **SUM** (Price \* Quantity) from OrderDetails **LEFT JOIN** Products **ON** OrderDetails.ProductID = Products.ProductID; |
| 5. | AVG () | Returns the average value of a numerical column.  **AVG** () or **AVG** ([ALL|DISTINCT] expression)  e.g.,  Select **AVG**(Price) from Products;  Select \* from Products WHERE price > (Select **AVG**(Price) from Products);  Select **AVG**(Price) AS AveragePrice, CategoryID **FROM** Products **GROUP BY** CategoryID; |

**SQL Joins**

* A JOIN clause is used to combine 2 or more tables, based on related columns between them.

|  |
| --- |
| **INNER JOIN or JOIN**   * Returns records only that have matching values in both tables. * **Select** columnNames **FROM** table1 **INNER JOIN / JOIN** table2 **ON** table1.column = table2.column   Example:   * **Select** Products.ProductID, Products.ProductName, Categories.CategoryName **FROM** Products **INNER JOIN** Categories **ON** Products.CategoryID = Categories.CategoryID; * **Select** Products.ProductID, Products.ProductName, Categories.CategoryName **FROM** Products **JOIN** Categories **ON** Products.CategoryID = Categories.CategoryID; |
| **LEFT (OUTER) JOIN**   * Returns all records from the left table, & the matched records from the right table. * The result is 0 records from right side if there is no match; * **Select** columnNames **FROM** table1 **LEFT JOIN** table2 **ON** table1.column = table2.column   Example:   * **Select** Customers.CustomerName, Orders.OrderID **FROM** Customers **LEFT JOIN** Orders **ON** Customers.CustomerID = Orders.CustomerID **ORDER BY** Customers.CustomerName; |
| **RIGHT (OUTER) JOIN**   * Returns all records from the right table, & the matched records from the left table. * The result is 0 records from the left side, if there is no match. * **Select** columnNames **FROM** table1 **RIGHT JOIN** table2 **ON** table1.column = table2.column   Example:   * **Select** Orders.OrderID, Employees.LastName, Employees.FirstName **FROM** Orders **RIGHT JOIN** Employees **ON** Orders.EmployeeID = Employees.EmployeeID **ORDER BY** Orders.OrderID; |
| **FULL (OUTER) JOIN or FULL JOIN**   * Returns all records when there is a match in either left or right table. * **Select** columnNames **FROM** table1 **FULL OUTER JOIN** table2 **ON** table1.column = table2.column **WHERE** condition;   Example:   * **Select** Customers.CustomerName, Orders.OrderID **FROM** Customers **FULL OUTER JOIN** Orders **ON** Customers.CustomerID = Orders. |



|  |
| --- |
| **Self-JOIN**   * A Self Join is a regular join, but the table is joined with itself. * Select columns FROM table1 T1, table1 T2 WHERE condition;   Example:   * **Select** A.CustomerName **AS** CustomerName1, B.CustomerName **AS** CustomerName2, A.City **FROM**   Customer A, Customer B **WHERE** A.CustomerID <> B.CustomerID AND A.City = B.City **ORDER BY** A.City; |

**SQL UNION, UNION ALL Operator**

* UNION operator is used to combine the result-set of 2 or more SELECT statements.
* UNION selects only distinct values. Use UNION ALL to also select duplicate values.
* Every SELECT statement within UNION must have the same number of columns.
* The Columns must also have similar data types.
* The Columns in every SELECT statement must also be in the same order.
* Syntax: **SELECT** column **FROM** table1 **UNION** **SELECT** column **FROM** table2;
* Syntax: **SELECT** column FROM table1 **UNION ALL** **SELECT** column **FROM** table2;

**Example:**

* **Select** City **FROM** Customers **UNION** **Select** City **FROM** Suppliers **ORDER BY** City;
* **Select** City **FROM** Customers **UNION** **ALL** **Select** City **FROM** Suppliers **ORDER BY** City;

**SQL Constraints**

* SQL constraints are used to specify rules for the data in a table.
* Constraints are used to limit the type of data that can go into a table. This ensures the accuracy & reliability of the data in the table.
* If there is any violation between the constraints & the data action, the action is aborted.
* Constraints can be column level or table level.
* Syntax: CREATE TABLE tableName (col1 datatype constraint, col2 datatype constraint…);

|  |  |  |
| --- | --- | --- |
| NOT NULL   * Ensures that a column can’t have a NULL value.   **Example:**   * CREATE TABLE Persons (ID int NOT NULL, lastName varchar (255) NOT NULL, Age int); * ALTER TABLE Persons ALTER/MODIFY COLUMN Age int NOT NULL; | | |
| UNIQUE   * Ensures that all values in a column are different. * Both the UNIQUE & PRIMARY KEY constraints provide a guarantee for uniqueness for a column or set of columns. A PRIMARY KEY constraint automatically has a UNIQUE constraint. * We can have many UNIQUE constraints per table, but only one PRIMARY KEY constraint per table.   **Example:**   * CREATE TABLE Persons (ID int NOT NULL, Age int, **UNIQUE**(ID)); * CREATE TABLE Persons (ID int NOT NULL, lastName varchar (255) NOT NULL, Age int, **CONSTRAINT** UC\_Person UNIQUE (ID, LastName)); * ALTER TABLE Persons ADD **UNIQUE** (ID); * ALTER TABLE Persons ADD **CONSTRAINT** UC\_Person **UNIQUE** (ID, LastName); * ALTER TABLE Persons DROP INDEX UC\_Person // MYSQL * ALTER TABLE Persons DROP CONSTRAINT UC\_Person; | | |
| PRIMARY KEY   * A combination of a NOT NULL & UNIQUE. Uniquely identifies each row in a table. * Primary keys must contain UNIQUE values, and can’t contain NULL values. A table can have only ONE primary key; and in the table, this primary key can consist of single or multiple columns (fields).   **Example:**   * CREATE TABLE Persons (ID int NOT NULL PRIMARY KEY, Age int); * CREATE TABLE Persons (ID int NOT NULL, LastName varchar (255) NOT NULL, Age int, PRIMARY KEY(ID)); * CREATE TABLE Persons (ID int NOT NULL, LastName varchar (255) NOT NULL, Age int, CONSTRAINT PK\_Person PRIMARY KEY (ID, LastName)); * ALTER TABLE Persons ADD PRIMARY KEY (ID); * ALTER TABLE Persons ADD CONSTRAINT PK\_Person PRIMARY KEY (ID, LastName); * ALTER TABLE Persons DROP PRIMARY KEY; * ALTER TABLE Persons DROP CONSTRAINT PK\_Person; | | |
| FOREIGN KEY   * Prevents actions that would destroy links between tables. * A FOREIGN KEY is a field (or collection of fields) in one table, that reers to the PRIMARY KEY in another table. * Table with the Foreign key is called Child Table & the table with the Primary key is called the referenced or parent table.   **Example:**   * CREATE TABLE Orders (OrderID int NOT NULL, OrderNumber int NOT NULL, PersonID int, **PRIMARY KEY**(OrderID), **FOREIGN KEY**(PersonID) **REFERENCES** Persons (PersonID)); * CREATE TABLE Orders (OrderID int NOT NULL PRIMARY KEY, OrderNumber int NOT NULL,   PersonID int **FOREIGN KEY REFERENCES** Persons (PersonID));   * ALTER TABLE Orders **ADD FOREIGN KEY (**PersonID) **REFERENCES** Persons (PersonID); * ALTER TABLE Orders **ADD CONSTRAINT** FK\_PersonOrder **FOREIGN KEY** (PersonID) **REFERENCES** Persons (PersonID); * ALTER TABLE Orders DROP **FOREIGN** **KEY** FK\_PersonOrder; * ALTER TABLE Orders DROP **CONSTRAINT** FK\_PersonOrder; | | |
| 5. | CHECK | Ensures that the values in a column satisfies a specific condition. |
| 6. | DEFAULT | Sets a default value for a column if no value is specified. |
| 7. | CREATE INDEX | Used to create & retrieve data from the database very quickly. |

Various Types of KEYS

* SQL supports various types of keys:

1. Primary Key
2. Candidate Key
3. Unique Key
4. Composite Key
5. Super Key
6. Alternate Key
7. Foreign Key

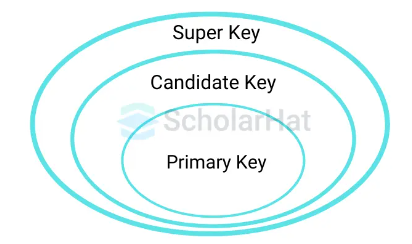
1. Candidate key

* A Candidate key is a set of one or more fields/columns that can identify a record uniquely in a table. There can be multiple candidate keys in one table.
* Each candidate key can work as Primary key. The value for the candidate key is always Unique & non-null for all the tuples (Record) types.
* One thing to remember is Every table should have at least one Candidate key but there can be more than one candidate key in a table. It is a super key with no repeated data.

2. Primary key

* A primary key is a set of one or more fields/columns of a table that uniquely identify a record in a database table. It can identify only one tuple (a record) at a time.
* It can’t accept null or duplicate values.
* Only one candidate key can be the Primary key.
* Out of all the candidate keys that can be possible or created for the specific table, there should be only one key that can be used to retrieve unique tuples from the table.

3. Super key

* A super key is a set of one or more than one key that can be used to identify a record uniquely in a table. E.g., Primary key, Unique key, & Alternate key are a subset of Super keys.
* Super key can contain multiple attributes that might not be able to identify tuples in a table independently, but when grouped with certain keys, they can identify tuples uniquely. It supports NULL values.
* Adding zero or more attributes to the candidate key generates the super key.
* A candidate key is a super key but vice versa is not true.

4. Alternate Key

* An Alternate key is a key that can work as a primary key. It is a candidate key that currently is not a primary key. It is also called Secondary key.
* The alternate key is a set of one or more than one column in a table that can uniquely identify each row in that table. Every table can have multiple options for a primary key, but only one column can be set as the primary key. All the keys which are not primary keys are called the Alternate keys.
* It maintains data integrity by providing a backup option for uniquely identifying records. While Primary key is the direct path to distinctiveness, the alternate key is a secondary assurance. This secondary assurance becomes crucial when dealing with scenarios where the primary key might not be feasible due to complexity, length, or other considerations.

5. Composite / Compound key

* A composite key is a combination of more than one field/column of a table. It can be a Candidate key or a Primary key.
* It gets its importance when it’s not possible for a single column/attribute to uniquely identify all the records of a table. It acts as a primary key if there is no primary key in a table.
* To create a composite key, we need to define multiple columns as primary or unique.

6. Unique key

* Unique key can be a column or set of columns that can be used to uniquely identify the tuple from the database. One or more fields can be declared as a unique key.
* **The unique key column can also hold the NULL value. The** use of unique keys improves the performance of data retrieval. It makes searching for records from the database must faster & efficient.

7. Foreign key

* A foreign key is an attribute that is a Primary key in its parent table but is included as an attribute in another host table.
* The relation that is being referenced is called the **Referenced Relation** & the corresponding attribute is called the **Referenced Attribute**.
* Foreign key may have duplicate & NULL values if it’s defined to accept NULL values.

|  |  |  |
| --- | --- | --- |
| **Key Type** | **Purpose** | **Characteristics** |
| Primary key | Used to uniquely identify a row in a table | Can’t be NULL, & must be a unique one per table. |
| Foreign key | Used to maintain referential integrity b/w tables | It can’t be NULL |
| Composite key | Used to uniquely identify a row when a single column is not sufficient. | It is a combination of columns; however, they must be unique. |
| Unique key | Used to prevent duplicate values in a column. | It can be NULL. |
| Candidate key | Used to identify potential Primary key | It can be unique & can uniquely identify each row in a table. |
| Super Key | Used to uniquely identify rows in a broad sense. | It can contain additional non – unique columns. |

**Practice Questions**

|  |
| --- |
| Q. Show first name, last name, and gender of patients whose gender is 'M'  Ans: SELECT first\_name, last\_name, gender FROM patients WHERE gender = 'M' |
| Q. Show first name and last name of patients who does not have allergies. (null)  Ans: SELECT first\_name, last\_name FROM patients WHERE allergies IS null; |
| Q. Show first name of patients that start with the letter 'C'  Ans: SELECT first\_name FROM patients WHERE first\_name like 'C%'  Ans: SELECT first\_name FROM patients WHERE substring(first\_name, 1, 1) = 'C' |
| Q. Show first name and last name of patients that weight within the range of 100 to 120 (inclusive)  Ans: SELECT first\_name, last\_name FROM patients WHERE weight BETWEEN 100 AND 120;  Ans: SELECT first\_name, last\_name FROM patients WHERE weight >= 100 AND weight <= 120; |
| Q. Update the patients table for the allergies column. If the patient's allergies is null then replace it with 'NKA'  Ans: UPDATE patients SET allergies = 'NKA' WHERE allergies IS NULL; |
| Q. Show first name and last name concatinated into one column to show their full name.  Ans: SELECT CONCAT (first\_name, ' ', last\_name) AS full\_name FROM patients;  Ans: SELECT first\_name || ' ' || last\_name FROM patients; |
| Q. Show first name, last name, and the **full** province name of each patient.  Ans: SELECT first\_name, last\_name, province\_name FROM patients **JOIN** province\_names **ON** province\_names.province\_id = patients.province\_id;  Ans: SELECT first\_name, last\_name, province\_name FROM patients, province\_names **ON** province\_names.province\_id = patients.province\_id; |
| Q. Show how many patients have a birth\_date with 2010 as the birth year.  Ans: SELECT COUNT (\*) AS total\_patients FROM patients WHERE YEAR (birth\_date) = 2010;  Ans: SELECT count(first\_name) AS total\_patients FROM patients WHERE birth\_date >= '2010-01-01' AND birth\_date <= '2010-12-31' |
| Q. Show the first\_name, last\_name, and height of the patient with the greatest height.  Ans: SELECT first\_name, last\_name, MAX(height) AS height FROM patients;  Ans: SELECT first\_name, last\_name, height FROM patients WHERE height = (SELECT max(height) FROM patients) |
| Q. Show all columns for patients who have one of the following patient\_ids: 1,45,534,879,1000  Ans: SELECT \* FROM patients WHERE patient\_id IN (1, 45, 534, 879, 1000); |
| Q. Show the patient id and the total number of admissions for patient\_id 579.  Ans: SELECT patient\_id, COUNT (\*) AS total\_admissions FROM admissions WHERE patient\_id = 579; |
| Q. Based on the cities that our patients live in, show unique cities that are in province\_id 'NS'?  Ans: SELECT DISTINCT (city) AS unique\_cities FROM patients WHERE province\_id = 'NS';  Ans: SELECT city **FROM** patients **GROUP BY** city **HAVING** province\_id = 'NS'; |
| Q. Show unique birth years from patients and order them by ascending. (M)  Ans: SELECT DISTINCT YEAR(birth\_date) AS birth\_year FROM patients ORDER BY birth\_year;  Ans: SELECT year(birth\_date) FROM patients GROUP BY year(birth\_date)  Ans: SELECT year(birth\_date) AS birth\_year FROM patients GROUP BY birth\_year  Ans: SELECT year(birth\_date) AS birth\_year FROM patients GROUP BY birth\_year ORDER BY birth\_year |
| Q. Show unique first names from the patients table which only occurs once in the list. (M)  Ans: SELECT first\_name FROM patients GROUP BY first\_name HAVING COUNT(first\_name) = 1  Ans: SELECT first\_name FROM **(**SELECT first\_name, count(first\_name) AS occurrencies **FROM** patients **GROUP BY** first\_name**)** WHERE occurrencies = 1 |
| Q. Show patient\_id and first\_name from patients where their first\_name start and ends with 's' and is at least 6 characters long.  Ans: SELECT patient\_id, first\_name FROM patients WHERE first\_name LIKE 's\_\_\_\_%s';  Ans: SELECT patient\_id, first\_name FROM patients WHERE first\_name LIKE 's%s' AND len(first\_name) >= 6;  Ans: SELECT patient\_id, first\_name FROM patients WHERE first\_name like 's%' and first\_name like '%s' and len(first\_name) >= 6; |
| Q. Show patient\_id, first\_name, last\_name from patients whos diagnosis is 'Dementia'. Primary diagnosis is stored in the admissions table.  Ans: SELECT patients.patient\_id, first\_name, last\_name FROM patients JOIN admissions ON  admissions.patient\_id = patients.patient\_id WHERE diagnosis = 'Dementia';  Ans: SELECT patient\_id, first\_name, last\_name FROM patients WHERE patient\_id IN (SELECT patient\_id FROM admissions WHERE diagnosis = 'Dementia'); |
| Q. Display every patient's first\_name. Order the list by the length of each name and then by alphabetically.  Ans: SELECT first\_name FROM patients order by len(first\_name), first\_name; |
| Q. |