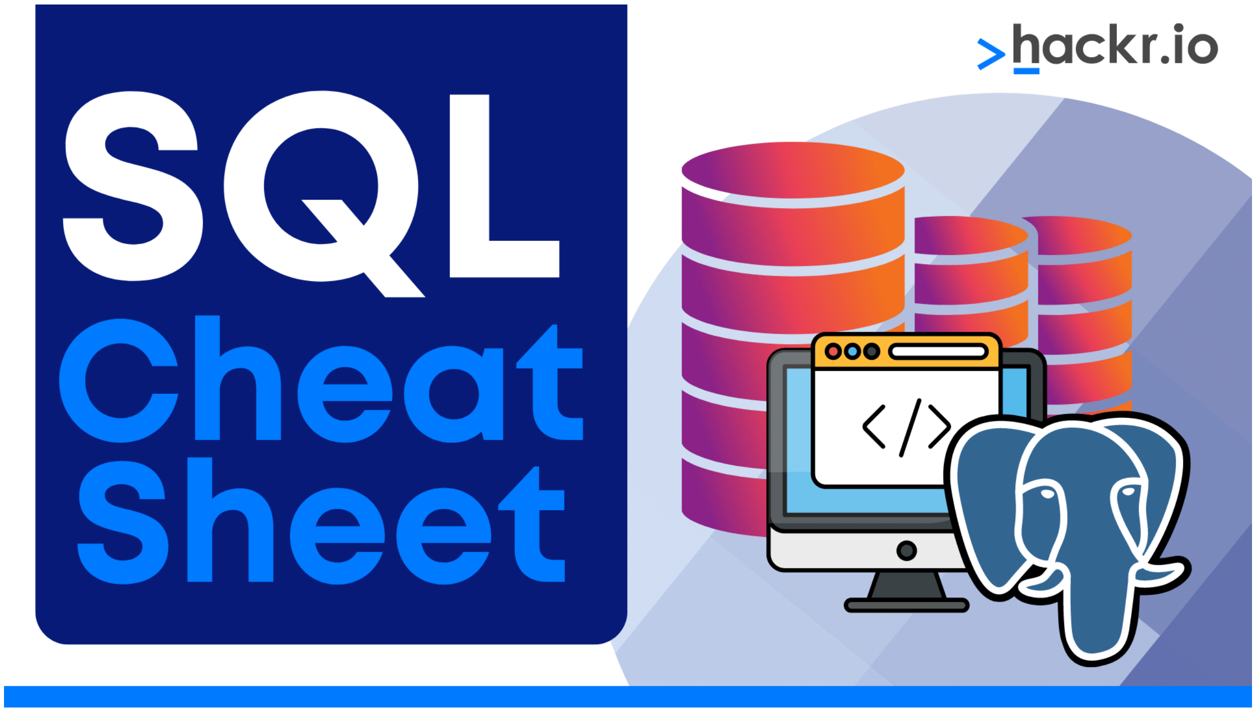

SQL Cheat Sheet

1. What is SQL? Why Do We Need It?

SQL is a database language that's used to query and manipulate data in a database, while also giving you an efficient and convenient environment for database management.

We can group commands into the following categories of **SQL statements:**

Data Definition Language (DDL) Commands

- **CREATE:** creates a new database object, such as a table
- **ALTER:** used to modify a database object
- **DROP:** used to delete database objects

Data Manipulation Language (DML) Commands

- **INSERT:** used to insert a new row (record) in a database table
- **UPDATE:** used to modify an existing row (record) in a database table
- **DELETE:** used to delete a row (record) from a database table

Data Control Language (DCL) Commands

- **GRANT:** used to assign user permissions to access database objects
- **REVOKE:** used to remove previously granted user permissions for accessing database objects

Data Query Language (DQL) Commands

- **SELECT:** used to select and return data (query) from a database

Data Transfer Language (DTL) Commands

- **COMMIT:** used to save a transaction within the database permanently
- **ROLLBACK:** restores a database to the last committed state

2. SQL Data Types

Data types specify the type of data that an object can contain, such as numeric data or character data. We need to choose a data type to match the data that will be stored using the following list of **essential pre-defined data types**:

Data Type	Used to Store
int	Integer data (exact numeric)
smallint	Integer data (exact numeric)
tinyint	Integer data (exact numeric)
bigint	Integer data (exact numeric)

decimal	Numeric data type with a fixed precision and scale (exact numeric)
numeric	Numeric data type with a fixed precision and scale (exact numeric)
float	Floating precision data (approximate numeric)
money	Monetary (currency) data
datetime	Date and time data
char(n)	Fixed length character data
varchar(n)	Variable length character data
text	Character string
bit	Integer data that is a 0 or 1 (binary)
image	Variable length binary data to store image files
real	Floating precision number (approximate numeric)
binary	Fixed length binary data
cursor	Cursor reference
sql_variant	Allows a column to store varying data types
timestamp	Unique database that is updated every time a row is inserted or updated
table	Temporary set of rows returned after running a table-valued function (TVF)

xml	Stores xml data

3. Managing Tables

After we’ve created a database, the next step is to create and subsequently manage a database table using a range of our **DDL commands**.

Create a Table

A table can be created with the **CREATE TABLE** statement.

Syntax for **CREATE TABLE:**

```
CREATE TABLE table_name
(
  col_name1 data_type,
  col_name2 data_type,
  col_name3 data_type,
  …
);
```

Example: Create a table named EmployeeLeave within the Human Resource schema and with the following attributes.

Columns	Data Type	Checks
EmployeeID	int	NOT NULL
LeaveStartDate	date	NOT NULL
LeaveEndDate	date	NOT NULL
LeaveReason	varchar(100)	NOT NULL
LeaveType	char(2)	NOT NULL

```
CREATE TABLE HumanResources.EmployeeLeave
(
  EmployeeID INT NOT NULL,
  LeaveStartDate DATETIME NOT NULL,
  LeaveEndDate DATETIME NOT NULL,
  LeaveReason VARCHAR(100),
  LeaveType CHAR(2) NOT NULL
);
```

SQL Table Constraints

Constraints define rules that ensure consistency and correctness of data. A **CONSTRAINT** can be created with either of the following approaches.

```
CREATE TABLE statement;
ALTER TABLE statement;
```

```
CREATE TABLE table_name
( Col_name data_type,
  CONSTRAINT constraint_name constraint_type col_name(s)
);
```

The following list details the various options for **Constraints:**

Constraint	Description	Syntax

Primary key	Columns or columns that uniquely identifies each row in a table.	`CREATE TABLE table_name`
		`(`
		`  col_name data_type,`
		`  CONSTRAINT constraint_name`
		`  PRIMARY KEY (col_name(s))`
		`);`

Unique key	Enforces uniqueness on non-primary key columns.	`CREATE TABLE table_name`
		`(`
		`  col_name data_type,`
		`  CONSTRAINT constraint_name`
		`  UNIQUE KEY (col_name(s))`
		`);`

Foreign key	Links two tables (parent & child), and ensures the child table's foreign key is present as the primary key in the parent before inserting data.	`CREATE TABLE table_name`
		`(`
		`  col_name data_type,`
		`  CONSTRAINT constraint_name`
		`  FOREIGN KEY (col_name)`
		`  REFERENCES`
		`  table_name(col_name)`
		`);`

Check	Enforce domain integrity by restricting values that can be inserted into a column.	`CREATE TABLE table_name`
		`(`
		`  col_name data_type,`
		`  CONSTRAINT constraint_name`
		`  CHECK (expression)`
		`);`

Modifying a Table

We can use the **ALTER TABLE** statement to **modify a table when:**

1. Adding a column
2. Altering a column's data type
3. Adding or removing constraints

Syntax for **ALTER TABLE:**

```
ALTER TABLE table_name
ADD column_name data_type;
```

```
ALTER TABLE table_name
DROP COLUMN column_name;
```

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

Renaming a Table

A table can be renamed with the **RENAME TABLE statement:**

```
RENAME TABLE old_table_name TO new_table_name;
```

Dropping a Table vs. Truncating a Table

A table can be dropped or deleted by using the **DROP TABLE statement:**

```
DROP TABLE table_name;
```

The contents of a table can be deleted (without deleting the table) by using the **TRUNCATE TABLE statement:**

```
TRUNCATE TABLE table_name;
```

4. Manipulating Data

Database tables are rarely static and we often need to add new data, change existing data, or remove data using our **DML commands**.

Storing Data in a Table

Data can be added to a table with the **INSERT** statement.

Syntax for **INSERT:**

```
INSERT INTO table_name ( col_name1, col_name2, col_name3… )
VALUES ( value1, value2, value3… );
```

Example: Inserting data into the Student table.

```
INSERT INTO Student ( StudentID, FirstName, LastName, Marks )
VALUES ( ‘101’, ’John’, ’Ray’, ’78’ );
```

Example: Inserting multiple rows of data into the Student table.

```
INSERT INTO Student
VALUES ( 101, ’John’, ’Ray’, 78 ),
   ( 102, ‘Steve’, ’Jobs’, 89 ),
   ( 103, ‘Ben’, ’Matt’, 77 ),
   ( 104, ‘Ron’, ’Neil’, 65 ),
   ( 105, ‘Andy’, ’Clifton’, 65 ),
   ( 106, ‘Park’, ’Jin’, 90 );
```

Syntax for copying data from one table to another with the **INSERT statement:**

```
INSERT INTO table_name2
SELECT * FROM table_name1
WHERE [condition];
```

Updating Data in a Table

Data can be updated in a table with the **UPDATE** statement.

Syntax for **UPDATE:**

```
UPDATE table_name
SET col_name1 = value1, col_name2 = value2…
WHERE condition;
```

Example: Update the value in the Marks column to ‘85’ when FirstName equals ‘Andy’

```
UPDATE table_name
SET Marks = 85
WHERE FirstName = ‘Andy’;
```

Deleting Data from a Table

A row can be deleted with the **DELETE** statement.

Syntax for **DELETE:**

```
DELETE FROM table_name
WHERE condition;
```

```
DELETE FROM Student
WHERE StudentID = '103';
```

Remove all rows (records) from a table without deleting the table with **DELETE:**

```
DELETE FROM table_name;
```

5. Retrieving Data

We can display one or more columns when we retrieve data from a table. For example, we may want to view all of the details from the Employee table, or we may want to view a selection of particular columns.

Data can be retrieved from a database table(s) by using the **SELECT** statement.

Syntax for **SELECT:**

```
SELECT [ALL | DISTINCT] column_list
FROM [table_name | view_name]
WHERE condition;
```

Consider the data and schema for the Student table below.

StudentID	FirstName	LastName	Marks
101	John	Ray	78
102	Steve	Jobs	89
103	Ben	Matt	77
104	Ron	Neil	65
105	Andy	Clifton	65

106	Park	Jin	90

Retrieving Selected Rows

We can retrieve a selection of rows from a table with the **WHERE** clause and a **SELECT statement:**

```
SELECT * FROM Student
WHERE StudentID = 104;
```

Note: We should use the **HAVING** clause instead of **WHERE** with aggregate functions.

Comparison Operators

Comparison operators test for the similarity between two expressions.

Syntax for **Comparisons:**

```
SELECT column_list FROM table_name
WHERE expression1 [COMP_OPERATOR] expression2;
```

Example: Various comparison operations.

```
SELECT StudentID, Marks FROM Student
WHERE Marks = 90;

SELECT StudentID, Marks FROM Student
WHERE StudentID > 101;

SELECT StudentID, Marks FROM Student
WHERE Marks != 89;

SELECT StudentID, Marks FROM Student
WHERE Marks >= 50;
```

Logical Operators

Logical operators are used with **SELECT** statements to retrieve records based on one or more logical conditions. You can combine multiple logical operators to apply multiple search conditions.

Syntax for **Logical Operators:**

```
SELECT column_list FROM table_name
WHERE conditional_expression1 [LOGICAL_OPERATOR] conditional_expression2;
```

Types of Logical Operator

We can use a range of logical operators to filter our data selections.

Syntax for **Logical OR Operator:**

```
SELECT StudentID, Marks FROM Student
WHERE Marks = 40 OR Marks = 56 OR Marks = 65;
```

Syntax for **Logical AND Operator:**

```
SELECT StudentID, Marks FROM Student
WHERE Marks = 89 AND LastName = ‘Jones’;
```

Syntax for **Logical NOT Operator:**

```
SELECT StudentID, Marks FROM Student
WHERE NOT LastName = ‘Jobs’;
```

Range Operations

We can use **BETWEEN** and **NOT BETWEEN** statements to retrieve data based on a range.

Syntax for **Range Operations:**

```
SELECT column_name1, col_name2… FROM table_name
WHERE expression1 RANGE_OPERATOR expression2 [LOGICAL_OPERATOR expression3…];
```

Syntax for **BETWEEN:**

```
SELECT StudentID, Marks FROM Student
WHERE Marks BETWEEN 40 AND 70;
```

Syntax for **NOT BETWEEN:**

```
SELECT FirstName, Marks FROM Student
WHERE Marks NOT BETWEEN 40 AND 50;
```

Retrieve Records That Match a Pattern

You can use the **LIKE** statement to fetch data from a table if it matches a specific string pattern. String patterns can be exact or they can make use of the **'%' and '_' wildcard** symbols.

Syntax for **LIKE with '%':**

```
SELECT * FROM Student
WHERE FirstName LIKE 'Ro%';
```

Syntax for **LIKE with '_':**

```
SELECT *FROM Student
WHERE FirstName LIKE '_e';
```

Displaying Data in a Sequence

We can display retrieved data in a specific order (ascending or descending) with **ORDER BY:**

```
SELECT StudentID, LastName FROM Student
ORDER BY Marks DESC;
```

Displaying Data Without Duplication

The **DISTINCT** keyword can be used to eliminate rows with duplicate values in a particular column.

Syntax for **DISTINCT:**

```
SELECT [ALL] DISTINCT col_names FROM table_name
WHERE search_condition;
```

```
SELECT DISTINCT Marks FROM Student
WHERE LastName LIKE 'o%';
```

6. SQL JOINS

Joins are used to retrieve data from more than one table where the results are 'joined' into a combined return data set. Two or more tables can be joined based on a common attribute.

Consider two database tables, Employees and EmployeeSalary, which we'll use to demonstrate joins.

EmployeeID (PK)	**FirstName**	**LastName**	**Title**
1001	Ron	Brent	Developer
1002	Alex	Matt	Manager
1003	Ray	Maxi	Tester
1004	August	Berg	Quality

EmployeeID (FK)	**Department**	**Salary**
1001	Application	65000
1002	Digital Marketing	75000
1003	Web	45000

1004	Software Tools	68000

Types of JOIN

The two main types of join are an **INNER JOIN** and an **OUTER JOIN.**

Inner JOIN

An inner join retrieves records from multiple tables when a comparison operation returns true for a common column. This can return all columns from both tables, or a set of selected columns.

Syntax for **INNER JOIN:**

```
SELECT table1.column_name1, table2.colomn_name2,…
  FROM table1
  INNER JOIN table2
  ON table1.column_name = table2.column_name;
```

Example: Inner join on Employees & EmployeeSalary tables.

```
SELECT Employees.LastName, Employees.Title, EmployeeSalary.salary,
  FROM Employees
  INNER JOIN EmployeeSalary
  ON Employees.EmployeeID = EmployeeSalary.EmployeeID;
```

Outer JOIN

An outer join displays the following **combined data set:**

- Every row from one of the tables (depends on LEFT or RIGHT join)
- Rows from one table that meets a given condition

An outer join will display **NULL** for columns where it does not find a matching record.

Syntax for **OUTER JOIN:**

```
SELECT table1.column_name1, table2.colomn_name2,… FROM table1
  [LEFT|RIGHT|FULL]OUTER JOIN table2
  ON table1.column_name = table2.column_name;
```

LEFT OUTER JOIN: every row from the 'left' table (left of the LEFT OUTER JOIN keyword) is returned, and matching rows from the 'right' table are returned.

Example: Left outer JOIN.

```
SELECT Employees.LastName, Employees.Title, EmployeeSalary.salary
  FROM Employees
  LEFT OUTER JOIN EmployeeSalary
  ON Employees.EmployeeID = EmployeeSalary.EmployeeID;
```

RIGHT OUTER JOIN: every row from the 'right' table (right of the RIGHT OUTER JOIN keyword) is returned, and matching rows from the 'left' table are returned.

Example: Right outer JOIN.

```
SELECT Employees.LastName, Employees.Title, EmployeeSalary.salary
  FROM Employees
  RIGHT OUTER JOIN EmployeeSalary
  ON Employees.EmployeeID = EmployeeSalary.EmployeeID;
```

FULL OUTER JOIN: returns all the matching and non-matching rows from both tables, with each row being displayed no more than once.

Example: Full outer JOIN.

```
SELECT Employees.LastName, Employees.Title, EmployeeSalary.salary
  FROM Employees
  FULL OUTER JOIN EmployeeSalary
  ON Employees.EmployeeID = EmployeeSalary.EmployeeID;
```

Cross JOIN

Also known as the **Cartesian Product**, a **CROSS JOIN** between two tables (A and B) 'joins' each row from table A with each row in table B, forming 'pairs' of rows. The joined dataset contains 'combinations' of row 'pairs' from tables A and B.

The row count in the joined data set is equal to the number of rows in table A multiplied by the number of rows in table B.

Syntax for **CROSS JOIN:**

```
SELECT col_1, col_2 FROM table1
CROSS JOIN table2;
```

Equi JOIN

An **EQUI JOIN** is one which uses an **EQUALITY** condition for the table keys in a JOIN operation. This means that INNER and OUTER JOINS can be EQUI JOINS if the conditional clause is an equality.

Self JOIN

A **SELF JOIN** is when you join a table with itself. This is useful when you want to query and return correlatory information between rows in a single table. This is helpful when there is a ‘parent’ and ‘child’ relationship between rows in the same table.

Example: if the Employees table contained references that links employees to supervisors (who are also employees in the same table).

To prevent issues with ambiguity, it’s important to use aliases for each table reference when performing a SELF JOIN.

Syntax for **SELF JOIN:**

```
SELECT t1.col1 AS “Column 1”, t2.col2 AS “Column 2”
FROM table1 AS t1
JOIN table1 AS t2
WHERE condition;
```

7. SQL Subqueries

An SQL statement that is placed within another SQL statement is a **subquery**.

Subqueries are nested inside WHERE, HAVING or FROM clauses for SELECT, INSERT, UPDATE, and DELETE statements.

- **Outer Query:** represents the **main query**

- **Inner Query:** represents the **subquery**

Using the IN Keyword

We can use the **IN** keyword as a logical operator to filter data for a main query (outer query) against a list of subquery results. This because a subquery will be evaluated first due to inner nest position. This filtering is part of the main query's conditional clause.

Example: run a subquery with a condition to return a data set. The subquery results then become part of the main query's conditional clause. We can then use the **IN** keyword to filter main query results against subquery results for a particular column(s).

Syntax for **IN keyword:**

```
SELECT column_1 FROM table_name
WHERE column_2 [NOT] IN
  ( SELECT column_2
    FROM table_name [WHERE conditional_expression] );
```

Using the EXISTS Keyword

We can use the **EXISTS** keyword as a type of logical operator to check whether a subquery returns a set of records. This means that the operator will return **TRUE** if the evaluated subquery returns any rows that match the subquery statement.

We can also use **EXISTS** to filter subquery results based on any provided conditions. You can think of it like a conditional 'membership' check for any data that is processed by the subquery statement.

Syntax for **EXISTS keyword:**

```
SELECT column FROM table_name
WHERE EXISTS
  ( SELECT column_name FROM table_name [WHERE condition] );
```

Using Nested Subqueries

Any individual subquery can also contain one or more additionally **nested subqueries**. This is similar to nesting conditional statements in traditional programming, which means that queries will be evaluated from the innermost level working outwards.

We use nested subqueries when the condition of one query is dependent on the result of another, which in turn, may also be dependent on the result of another etc.

Syntax for **Nested Subqueries:**

```
SELECT col_name FROM table_name
WHERE col_name(s) [LOGICAL | CONDITIONAL | COMPARISON OPERATOR]
   ( SELECT col_name(s) FROM table_name
     WHERE col_name(s) [LOGICAL | CONDITIONAL | COMPARISON OPERATOR]
       ( SELECT col_name(s) FROM table_name
         WHERE [condition] )
  );
```

Correlated Subquery

A correlated subquery is a special type of subquery that uses data from the table referenced in the outer query as part of its own evaluation.

8. Using Functions to Customize a Result Set

Various built-in functions can be used to customize a result set.

Syntax for **Functions:**

```
SELECT function_name (parameters);
```

Using String Functions

When our result set contains strings that are **char** and **varchar** data types, we can manipulate these string values by using **string functions**:

Function Name	Example
left	`SELECT left('RICHARD', 4);`

len	`SELECT len(‘RICHARD’);`
lower	`SELECT lower(‘RICHARD’);`
reverse	`SELECT reverse(‘ACTION’);`
right	`SELECT right(‘RICHARD’, 4);`
space	`SELECT ‘RICHARD’ + space(2) + ‘HILL’;`
str	`SELECT str(123.45, 6, 2);`
substring	`SELECT substring(Weather’, 2, 2);`
upper	`SELECT upper(‘RICHARD’);`

Using Date Functions

When our result set contains date and time data, we may want to manipulate it to extract the day, month, year, or time, and we may also want to parse date-like data into a datetime data type. We can do this by using **date functions:**

Function Name	Parameters	Description
dateadd	(date part, number, date)	Adds the ‘number’ of date parts to the date
datediff	(date part, date1, date2)	Calculates the ‘number’ of date parts between two dates
Datename	(date part, date)	Returns the date part from a given date as a character value

datepart	(date part, date)	Returns the date part from a given date as an integer value
getdate	0	Returns the current date and time
day	(date)	Returns an integer to represent the day for a given date
month	(date)	Returns an integer to represents the month for a given date
year	(date)	Returns an integer to represents the year for a given date

Using Mathematical Functions

We can manipulate numeric data types within our result set by using **mathematical functions:**

Function Name	**Parameters**	**Description**
abs	(numeric_expression)	Returns the absolute value
acos, asin, atan	(numeric_expression)	Returns the arc cos, sin, or tan angle in radians
cos, sin, tan, cot	(numeric_expression)	Returns the cos, sine, tan or cotangent in radians
degrees	(radians)	Returns an angle in degrees converted from radians
exp	(numeric_expression)	Returns the value of e raised to the power of a given number or expression

floor	(numeric_expression)	Returns the largest integer value less than or equal a given value
log	(numeric_expression)	Returns the natural logarithm of a given value
pi	0	Returns the constant value of pi which is 3.141592653589793…
power	(numeric_expression, y)	Returns the value of a numeric expression raised to to the power of y
radians	(degrees)	Returns an angle in radians converted from degrees
rand	([seed])	Returns a random float number between 0 and 1 inclusive
round	(number, precision)	Returns a rounded version of a given numeric value to a given integer value for precision
sign	(numeric_expression)	Returns the sign of a given value, which can be positive, negative or zero
sqrt	(numeric_expression)	Returns the square root of a given value

Using Ranking Functions

Ranking functions (also known as **window functions**) generate and return sequential numbers to represent a rank for each based on a given criteria. To rank records, we use the following **ranking functions:**

- **row_number() :** returns sequential numbers starting at 1, for each row in in a result set based on a given column

- **rank() :** returns the rank of each row in a result set based on specified criteria (can lead to duplicate rank values)
- **dense_rank() :** used when consecutive ranking values are needed for a given criteria (no duplicate rank values)

Each ranking function uses the **OVER** clause to specify the ranking criteria. Within this, we choose a column to use for assigning a rank along with the **ORDER BY** keyword to determine whether ranks should be applied based on ascending or descending values.

Using Aggregate Functions

Aggregate functions summarize values for a column or group of columns to produce a single (aggregated) value.

Syntax for **Aggregate Functions:**

```
SELECT AGG_FUNCTION( [ALL | DISTINCT] expression )
FROM table_name;
```

The table below summarizes the various **SQL aggregate functions:**

Function Name	Description
avg	Returns the average from a range of values in a given data set or expression. Can include **ALL** values or **DISTINCT** values
count	Returns the quantity (count) of values in a given data set or expression. Can include **ALL** values or **DISTINCT** values
min	Returns the lowest value in a given data set or expression

max	Returns the highest value in a given data set or expression
sum	Returns the sum of values in a given data set or expression. Can include **ALL** values or **DISTINCT** values

9. Grouping Data

We have the option to group data in our result set based on a specific criteria. We do this by using the optional **GROUP BY**, **COMPUTE**, **COMPUTE BY**, and **PIVOT** clauses with a **SELECT** statement.

GROUP BY Clause

When used without additional criteria, **GROUP BY** places data from a result set into unique groups. But when used with an aggregate function, we can summarize (aggregate) data into individual rows per group.

Syntax for **GROUP BY:**

```
SELECT column(s) FROM table_name
GROUP BY expression
[HAVING search_condition];
```

COMPUTE and COMPUTE BY Clause

We can use the **COMPUTE** clause with a **SELECT** statement and an **aggregate function** to generate summary rows as a separate result from our query. We can also use the optional **BY** keyword to calculate summary values on a column–by-column basis.

Syntax for **COMPUTE [BY]:**

```
SELECT column(s) FROM table_name
[ORDER BY column_name]
COMPUTE [BY column_name]  AGG_FUNCTION(column_name)
```

Note: support for this keyword was dropped by MS SQL Server in 2012.

PIVOT Clause

The **PIVOT** operator is used to transform unique rows into column headings. You can think of this as rotating or pivoting the data into a new 'pivot table' that contains the summary (aggregate) values for each rotated column. With this table, you can examine trends or summary values on a columnar basis.

Syntax for **PIVOT:**

```
SELECT * FROM table_name
PIVOT ( AGG_FUNCTION (value_column)
        FOR pivot_column
        IN column(s) )
AS pivot_table_alias;
```

10. The ACID Property

The term **ACID** stands for **Atomicity**, **Consistency**, **Isolation**, and **Durability**. These individual properties represent a standardized group that are required to ensure the reliable processing of database transactions.

Atomicity

The concept that an entire transaction must be processed fully, or not at all.

Consistency

The requirement for a database to be consistent (valid data types, constraints, etc) both before and after a transaction is completed.

Isolation

Transactions must be processed in isolation and they must not interfere with other transactions.

Durability

After a transaction has been started, it must be processed successfully. This applies even if there is a system failure soon after the transaction is started.

11. RDBMS

A **Relational Database Management System** (RDBMS) is a piece of software that allows you to perform database administration tasks on a relational database, including creating, reading, updating, and deleting data (**CRUD**).

Relational databases store collections of data via columns and rows in various tables. Each table can be related to others via common attributes in the form of Primary and Foreign Keys.