**Chapter 3: Introduction to SQL**

Table of Contents

[Create Table Construct 3](#_Toc66076588)

[Integrity Constraints 3](#_Toc66076589)

[Updates to Relations 4](#_Toc66076590)

[Basic Query Structure 5](#_Toc66076591)

[The Rename Operation 8](#_Toc66076592)

[String Operations 9](#_Toc66076593)

[Ordering Display of Tuples 10](#_Toc66076594)

[Where Clause Predicates 11](#_Toc66076595)

[Set Operations 12](#_Toc66076596)

[NULL Values 13](#_Toc66076597)

[Aggregate Functions 14](#_Toc66076598)

[Nested Subqueries 17](#_Toc66076599)

IBM first invented the Sequel language, which was later renamed the Structured Query Language, or SQL. SQL became widely used and it became necessary to follow some standard. Thus, we have several ANSI and ISO standards for SQL including SQL 86, SQL 89, SQL 92, SQL 1999 and SQL 2003. Commercial systems tend to follow the SQL 92 standard.

The SQL data definition language (DDL) allows specification of information about relations including the schema for relations, the domain of values associated with each attribute, integrity constraints and more.

Domain Types:

* char (n) – Character string of fixed length n.
* varchar (n) – Character string of variable length, with a maximum length of n.
* int – Integers
* smallint – Small integers
* numeric (p, d) – Numbers of length p, with d decimal places.
* real, double precision – Floating point and double precision floating point numbers with machine dependant precision
* float (n) – Floating point number with user specified precision of at least n digits.

## Create Table Construct

SQL relations are defined using the create table command:

CREATE TABLE instructor(  
ID CHAR(5),  
name VARCHAR(20) NOT NULL,  
dept\_name VARCHAR(20),  
salary NUMERIC(8, 2),  
PRIMARY KEY (ID),  
FOREIGN KEY (dept\_name) REFERENCES department);

SQL

Integrity Constraints:

* NOT NULL – ensures value cannot be null
* PRIMARY KEY – declares a primary key (automatically assigns NOT NULL)
* FOREIGN KEY – declares a foreign key referencing another relation

## Updates to Relations

* Insert record into a relation (notice single quotes around strings and not double quotes)

INSERT INTO instructor VALUES ('10211', 'Smith', 'Biology', 66000);

SQL

* Delete all records from a relation

DELETE FROM student;

SQL

* Drop entire table

DROP TABLE r;

SQL

* Add attributes to a relation (all existing records have null in place of new attribute)

ALTER TABLE r ADD name CHAR(20);

SQL

* Remove attribute from a relation (not supported by many databases)

ALTER TABLE r DROP name;

SQL

## Basic Query Structure

SELECT ID FROM students WHERE CGPA = 4.0;

SQL

The result of an SQL query is a relation. This is very important to remember.

The SELECT keyword returns a relation that has only the attributes selected. For example,

SELECT name FROM students;

SQL

will return all the names from all the records in the students relation. Remember that the code is case insensitive so writing name is the same as writing NAME or Name. The values however, are case sensitive.

SQL will allow duplicate values to exist in a relation as well as in query results. To eliminate duplicates, we use the DISTINCT keyword.

SELECT DISTINCT dept\_name FROM instructor;

SQL

The keyword ALL on the other hand, ensures duplicates are kept.

SELECT ALL dept\_name FROM instructor;

SQL

If we use SELECT \*, it means all attributes from a certain relation. If we use SELECT '437', it will select that particular value and no other attributes or records, and will not need a FROM clause. We can also give the column a name like SELECT '437' AS FOO. We can also select a literal with the FROM clause, like this:

SELECT 'A' FROM instructors;

SQL

which will return a relation with the number of rows that instructor had, with each cell having a value ‘A’.

We can also perform arithmetic operations with the select clause. Using this:

SELECT ID, name, salary/12 AS monthly\_salary FROM instructor;

SQL

will return a relation containing the particular attributes, except the values of the salary attributes will be divided by 12 and the attribute itself will be renamed to monthly\_salary.

The WHERE clause specifies conditions. Only records matching the condition will be returned. We can also combine conditions using AND, OR and NOT logical operators.

SELECT \* FROM instructors WHERE dept\_name = 'CSE' AND salary > 8000;

SQL

The FROM clause lists relations involved in a query. It essentially gets the cartesian product of the relations, while SELECT and WHERE do the searching.

SELECT \* FROM instructor, teaches;

SQL

will return the cartesian product of the two relations instructor and teaches with all attributes from both. If there are common attributes, like ID, they are renamed using the relation name like instructor.ID.

SELECT name, course\_id FROM instructor, teaches  
WHERE instructor.ID = teaches.ID;

SQL

For every instructor in the instructor relation, for records that have the same ID in the teaches relation, the name and course\_id for that instructor is the resulting relation.

The bit where we checked whether ID under instructor had the same value as ID under teaches using the WHERE clause is called joining. To make life easier, we can do something called a natural join like this:

SELECT name, course\_id FROM instructor NATURAL JOIN teaches;

SQL

This will do the same thing. However, we need to make sure that the matching attribute has the same name in both relations.

## The Rename Operation

As seen before, the AS keyword can be used to rename things in our resulting query. So, this makes perfect sense.

SELECT DISTINCT T.name  
FROM instructor AS T, instructor AS S  
WHERE T.salary > S.salary AND s.dept\_name = 'CSE'

SQL

This returns the names of all instructors that have a salary more than even one instructor in the CSE department.

The keyword AS can in fact be omitted. instructor T would work just fine.

## String Operations

The operator LIKE can be used to match patterns in strings.

SELECT name FROM instructor WHERE name LIKE '%dar%';

SQL

This returns any names that have “dar” in any part.

If we wanted to use the % character itself, it would have to be escaped like “100\%”.

Underscores indicate character. '\_ \_ \_' would return any string with exactly 3 characters.

There are other operations too like concatenation (using “||”), conversion to upper or lower case (using UPPER and LOWER respectively), finding string length, extracting substrings etc.

## Ordering Display of Tuples

We can set whether tuples will appear in ascending or descending order like this:

SELECT DISTINCT name FROM instructor ORDER BY name DESC;

SQL

If nothing was stated after name, the default ASC would be used.

We can also sort on multiple attributes, separating each with a comma.

## Where Clause Predicates

The BETWEEN operator allows us to search for data lying within a range.

SELECT name FROM instructor WHERE salary BETWEEN 90000 AND 100000;

SQL

Tuple Comparisons can also be done.

WHERE (instructor.ID, dept\_name) = (teaches.ID, 'Biology');

SQL

## Set Operations

We can use UNION, INTERSECTS and EXCEPT in queries like this:

(SELECT course\_id FROM section WHERE sem = 'Fall')  
UNION  
(SELECT course\_id FROM section WHERE sem = 'Spring');

SQL

This will return courses that are present in either Fall or Spring semesters. Note that it is not possible to perform a union on relations with no common attributes. Using INTERSECTS would return courses only present in both, and EXCEPT would return courses present in Fall, but not in Spring. Each of these operations automatically eliminates duplicates. To retain duplicates, we can use UNION ALL, INTERSECT ALL and EXCEPT ALL.

## NULL Values

Tuples can have NULL values, which indicate that the value is unknown or does not exist. Any arithmetic operations involving a NULL value results in NULL. The predicate IS NULL can be used to check for null values.

SELECT name FROM instructor WHERE salary IS NULL;

SQL

There is another possible value called UNKNOWN. Comparisons with NULL values returns UNKNOWN. Logic operations with UNKNOWN depend on what is used (UNKNOWN OR TRUE will give true for example). ‘P IS UNKNOWN’ checks for UNKNOWN values. Results of a WHERE clause that evaluate to UNKNOWN are treated as false.

## Aggregate Functions

Aggregate functions go over multiple tuples and return a single value. AVG, MIN, MAX, SUM and COUNT are all aggregate values.

SELECT AVG (salary) FROM instructor WHERE dept\_name = 'CSE';

SQL

The COUNT operator can be a bit misleading. It returns how many records there are that match the condition.

Remember that the WHERE clause is executed before any aggregate functions, so the query gets all instructors of the CSE department, before calculating the average salary over them.

The GROUP BY operator can be used alongside aggregate functions to group results.

SELECT dept\_name, AVG (salary) AS avg\_salary FROM instructor  
GROUP BY dept\_name;

SQL

This gives us the names and average salaries for each department. Attributes that appear under the SELECT clause alongside aggregate functions must be in the GROUP BY list. Otherwise, there will be an error.

The HAVING clause can be used to further reduce results.

SELECT dept\_name, AVG (salary) AS avg\_salary FROM instructor  
GROUP BY dept\_name HAVING AVG (salary) > 42000;

SQL

This returns names and average salaries of all departments for which the average salary is greater than 42000.

The HAVING clause is applied after the aggregate functions are executed (obviously) whereas the WHERE clause is executed before aggregate functions.

Note that we cannot use the alias we created for the average salary, avg\_salary, in the HAVING clause. We must use the original name, i.e. AVG (salary).

All aggregate functions except COUNT ignores tuples with NULL values. The only exception is when the collection has only NULL values, in which case COUNT returns 0 whereas the other functions return NULL.

Example 1:

SELECT Dept, AVG(CGPA) FROM Students GROUP BY Dept;

SQL

This will retrieve all the values under the CGPA attribute from the Students relations, find the average of each department, and show us the average CGPA for each department.

Example 2:

SELECT Dept, DeptSize, AVG(CGPA) AResult  
FROM Students S, Department D  
WHERE S.Dept = D.Name GROUP BY Dept;

SQL

This query should go to the Students relations, renamed S, and the Department relation, renamed D, find all records for which S.Dept is the same as D.Name, calculate the average CGPA, renamed AResult, for each department and show us a relation that holds this information, grouped by department. This is similar to the previous example, only that this uses two different relations and does a natural join. However, this query will cause an error. We have included the attribute DeptSize in the SELECT clause because we want that in our query result, but the resulting relation is not being grouped by this clause. SQL does not know what to do with this attribute. Fortunately, for this particular situation, we have a workaround. Since the value of DeptSize will be the same for every record of a single department, we can use the maximum or minimum value for this attribute and still get the same value. Since we are performing an aggregate function on that attribute, there will be no error.

SELECT Dept, MAX(DeptSize), AVG(CGPA) AResult  
FROM Students S, Department D  
WHERE S.Dept = D.Name GROUP BY Dept;

SQL

Example 3:

SELECT Dept, MAX(DeptSize), AVG(CGPA) AResult  
FROM Students S, Department D  
WHERE S.Dept = D.Name  
GROUP BY Dept HAVING AVG(CGPA) >= 3.75;

SQL

This is the same as example 2, except that we have added an extra HAVING clause. Thus, after everything else is done, the final results will simply not include the records for which the average CGPA is below the stated value. Notice that we cannot use the alias AResult we created here.

## Nested Subqueries

We previously stated that the result of every query in SQL is a relation itself, and we can perform further queries on those relations. This can be done under 2 positions.

1. The WHERE Clause

SELECT ID, Name, CGPA, FROM Students  
WHERE CGPA >= (SELECT AVG(CGPA) FROM Students);

SQL

This query attempts to show the ID, Name and CGPA attributes from the Students relation, but only those records that have a CGPA above a certain value. To find this certain value, the query must perform another query, which returns the average value of the CGPA attribute from all the records in the Students relation. The result is of course a single value, so this is valid.

2. The FROM Clause

SELECT ROWNUM, ID, Name  
FROM (SELECT ID, Name, CGPA FROM Students ORDER BY CGPA DESC)  
WHERE ROWNUM <= 3;

SQL

This is a little more complicated. Inside the nested subquery, we are creating a relation that gets the attributes ID, Name and CGPA from the Students relation, and arranges them in descending order based on the CGPA. We had to include the DESC keyword because the order would be ascending by default. From this query result, we had to select the ID and Name attributes and find just the top 3 records. Since the relation from the nested query was in descending order, we thus got the 3 records with the highest CGPA.

Notice that we had to include more attributes in the nested query here than we had to in the example we did with the WHERE clause. This is because we are telling SQL to find these attributes from the result of the nested subquery. If the nested subquery does not even have these attributes, we cannot find them there. The structure of our query is dependent on the nested subquery. The example with the WHERE clause was asking something different. It was creating its own relation, and deciding which records to display based on the results of the nested subquery. Thus, we do not need to get the extra attributes (I hope you kind of got that 🤨).

Also notice that we have used an attribute called ROWNUM. This is a built-in attribute for every relation and is literally the row number. It is not some value we can edit. If we were to sort some relation, the row numbers would still start from 1. It would not change. We do not normally need to display this, but we had to select it here since our WHERE clause is getting the top 3 rows.