Unit -2

Relational Query Language

- A query language is a language in which a user requests information from the database.
- These languages are usually on a level higher than that of a standard programming
  language.

Imperative query language

The user instructs the system to perform a specific sequence of operations on the database to compute the desired result; such languages usually have a notion of state
variables, which are updated in the course of the computation. Example lang

Functional query language

The computation is expressed as the evaluation of functions that may operate on data in the database or on the results of other functions; functions are side-effect free, and
they do not update the program state

Declarative query language:

The user describes the desired information without giving a specific sequence of steps or function calls for obtaining that information; the desired information is typically
described using some form of mathematical logic

- **Imperative** is like giving detailed instructions step by step.
- **Functional** is like using a tool (like a calculator) to perform operations without changing anything around.
- **Declarative** is like stating what you want without specifying how to get it, using more abstract descriptions.

Relational Algebra

Operators:

- Unary Operators:
  - Select: σ
  - Project: $\prod$
  - Rename: ρ
- Binary Operators:
  - Union: U
  - Intersection :$\cap$
  - Set difference: –
  - Cartesian product: X
  - Join: $\bowtie$

SELECT

The select operation selects tuples that satisfy a given predicate.

Query: $\sigma_{dept\_name="Physics"}(instructor)$

Result:

ID	name	dept_name	salary
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

- We allow comparisons to be used in the selection predicate.
  $=, \neq, >, \geq, <, \leq$
- We can combine several predicates into a larger predicate by using the connectives:
  $\wedge$ (**and**), $\vee$ (**or**), $\neg$ (**not**)
- Example: Find the instructors in Physics with a salary greater $90,000, we write:

  $\sigma_{dept\_name="Physics" \wedge salary > 90,000}(instructor)$
- The select predicate may include comparisons between two attributes.

Example: Find all departments whose name is the same as their building name:

$\sigma_{dept\_name=building}(department)$

PROJECT

A unary operation that returns its argument relation, with certain attributes left out.

The project operation removes any duplicate tuples.

Example:

Eliminate the *dept_name* attribute of *instructor*

Query:

$$\Pi_{ID,\ name,\ salary}\ (instructor)$$

Result:

ID	*name*	*salary*
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

PROJECT is *not* commutative

- $\pi_{\text{<list1>}}(\pi_{\text{<list2>}}(R)) = \pi_{\text{<list1>}}(R)$ as long as <list2> contains the attributes in <list1>

Example: To list each employee's first and last name and salary, the following is used

Employees

Surname	FirstName	Department	Head
Smith	Mary	Sales	De Rossi
Black	Lucy	Sales	De Rossi
Verdi	Mary	Personnel	Fox
Smith	Mark	Personnel	Fox

Query:

$\Pi_{\text{Surname,FirstName}}$ (EMPLOYEE)

Output:

$\pi_{\text{Surname, FirstName}}$(Employees)

Surname	FirstName
Smith	Mary
Black	Lucy
Verdi	Mary
Smith	Mark

- To retrieve the first name, last name, and salary of all employees who work in department number 5, we must apply a select and a project operation
- We can write a *single relational algebra expression* as follows:
  - $\pi_{FNAME, LNAME, SALARY}(\sigma_{DNO=5}(EMPLOYEE))$
- OR We can explicitly show the *sequence of operations*, giving a name to each intermediate relation:
  - DEP5_EMPS $\leftarrow \sigma_{DNO=5}(EMPLOYEE)$
  - RESULT $\leftarrow \pi_{FNAME, LNAME, SALARY}$ (DEP5_EMPS)

RENAME

The results of relational-algebra expressions do not have a name that we can use to
refer to them. The rename operator is provided for that purpose

The RENAME operator is denoted by ρ(rho)

Example of Renaming

Paternity

Father	Child
Adam	Cain
Adam	Abel
Abraham	Isaac
Abraham	Ishmael

$\rho_{Father \rightarrow Parent}(Paternity)$

Parent	Child
Adam	Cain
Adam	Abel
Abraham	Isaac
Abraham	Ishmael

- The textbook allows positions rather than attribute names, e.g., **1 → Parent**
- Textbook also allows renaming of the relation itself,e.g.,**Paternity,1→ Parenthood,Parent**

ASSIGNMENT OPERATOR

It is convenient at times to write a relational-algebra expression by assigning parts of it
to temporary relation variables.

- Example: Find all instructor in the “Physics” and Music department.

$Physics \leftarrow \sigma_{dept\_name="Physics"}(instructor)$

$Music \leftarrow \sigma_{dept\_name="Music"}(instructor)$

$Physics \cup Music$

$DEP5\_EMPS \leftarrow \sigma_{DNO=5}(EMPLOYEE)$

$RESULT \leftarrow \pi_{FNAME, LNAME, SALARY}(DEP5\_EMPS)$

Binary Operator

CARTESIAN PRODUCT

The Cartesian-product operation (denoted by X) allows us to combine information from
any two relations.

Instructor

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

Teaches

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2017
10101	CS-315	1	Spring	2018
10101	CS-347	1	Fall	2017
12121	FIN-201	1	Spring	2018
15151	MU-199	1	Spring	2018
22222	PHY-101	1	Fall	2017
32343	HIS-351	1	Spring	2018
45565	CS-101	1	Spring	2018
45565	CS-319	1	Spring	2018
76766	BIO-101	1	Summer	2017
76766	BIO-301	1	Summer	2018
83821	CS-190	1	Spring	2017
83821	CS-190	2	Spring	2017
83821	CS-319	2	Spring	2018
98345	EE-181	1	Spring	2017

Instructor X Teaches

instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2017
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2018
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2018
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
15151	Mozart	Music	40000	10101	CS-101	1	Fall	2017
15151	Mozart	Music	40000	10101	CS-315	1	Spring	2018
15151	Mozart	Music	40000	10101	CS-347	1	Fall	2017
15151	Mozart	Music	40000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
15151	Mozart	Music	40000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
22222	Einstein	Physics	95000	10101	CS-101	1	Fall	2017
22222	Einstein	Physics	95000	10101	CS-315	1	Spring	2018
22222	Einstein	Physics	95000	10101	CS-347	1	Fall	2017
22222	Einstein	Physics	95000	12121	FIN-201	1	Spring	2018
22222	Einstein	Physics	95000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

JOIN

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

- FEMALE_EMPS $\leftarrow \sigma_{SEX='F'}$(EMPLOYEE)
- EMPNAMES $\leftarrow \pi_{FNAME, LNAME, SSN}$ (FEMALE_EMPS)
- EMP_DEPENDENTS $\leftarrow$ EMPNAMES x DEPENDENT
- ACTUAL_DEPS $\leftarrow \sigma_{SSN=ESSN}$(EMP_DEPENDENTS)
- RESULT $\leftarrow \pi_{FNAME, LNAME, DEPENDENT\_NAME}$ (ACTUAL_DEPS)

- RESULT will now contain the name of female employees and their dependents

Query split-up: FEMALE_EMPS $\leftarrow \sigma_{SEX='F'}$(EMPLOYEE)

FEMALE_EMPS

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	1968-07-19	3321Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291Berry, Bellaire, TX	F	43000	888665555	4
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

EMPNAMES $\leftarrow \pi_{FNAME, LNAME, SSN}$ (FEMALE_EMPS)

EMPNAMES

Fname	Lname	Ssn
Alicia	Zelaya	999887777
Jennifer	Wallace	987654321
Joyce	English	453453453

Query split-up: $EMP\_DEPENDENTS \leftarrow EMPNAMES \times DEPENDENT$

EMP_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	...
Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	...
Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	...
Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28	...
Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04	...
Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	...
Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	...
Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	...
Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	...
Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...
Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	...
Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	...
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	...
Joyce	English	453453453	333445555	Alice	F	1986-04-05	...
Joyce	English	453453453	333445555	Theodore	M	1983-10-25	...
Joyce	English	453453453	333445555	Joy	F	1958-05-03	...
Joyce	English	453453453	987654321	Abner	M	1942-02-28	...
Joyce	English	453453453	123456789	Michael	M	1988-01-04	...
Joyce	English	453453453	123456789	Alice	F	1988-12-30	...
Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	...

Query split-up: $ACTUAL\_DEPS \leftarrow \sigma_{SSN=ESSN}(EMP\_DEPENDENTS)$

ACTUAL_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...

$RESULT \leftarrow \pi_{FNAME, LNAME, DEPENDENT\_NAME}(ACTUAL\_DEPS)$

RESULT

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

Find teachers who also instruct

$\sigma_{instructor.id = teaches.id}(instructor \times teaches)$

instructor.ID	name	dept_name	salary	teaches.ID	course_id	sec_id	semester	year
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
15151	Mozart	Music	40000	15151	MU-199	1	Spring	2018
22222	Einstein	Physics	95000	22222	PHY-101	1	Fall	2017
32343	El Said	History	60000	32343	HIS-351	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-101	1	Spring	2018
45565	Katz	Comp. Sci.	75000	45565	CS-319	1	Spring	2018
76766	Crick	Biology	72000	76766	BIO-101	1	Summer	2017
76766	Crick	Biology	72000	76766	BIO-301	1	Summer	2018
83821	Brandt	Comp. Sci.	92000	83821	CS-190	1	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-190	2	Spring	2017
83821	Brandt	Comp. Sci.	92000	83821	CS-319	2	Spring	2018
98345	Kim	Elec. Eng.	80000	98345	EE-181	1	Spring	2017

- Thus

$$\sigma_{instructor.id = teaches.id}(instructor \times teaches)$$

- Can equivalently be written as

$$instructor \bowtie_{Instructor.id = teaches.id} teaches.$$

Question

Example: Write an algebraic query to retrieve the details of employees along with their department id and name.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

$$[\pi_{\text{ ID,Name, Dno, Dnumber, Dname}}(\sigma_{\text{Dno = Dnumber}}(employee \bowtie department))]$$

Set Operations and Equivalent Queries

UNION

- The union operation allows us to combine two relations.
- The result of R U S, is a relation that includes all tuples that are either in R or in S or in both R and S
- **Duplicate tuples are eliminated**
- **R and S must have same number of attributes.**
- Each pair of corresponding attributes must be type compatible

For r U s to be valid.

- r, s must have the same arity (same number of attributes).
- The attribute domains must be compatible (example: 2nd column of r deals with the same type of values as does the 2nd column of s).

Find all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or in both

course_id	sec_id	semester	year	building	room_number	time_slot_id
BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A

$$\Pi_{course\_id}\,(\sigma_{semester=\text{“Fall”}\ \wedge\ year=2017}(section)) \cup \Pi_{course\_id}\,(\sigma_{semester=\text{“Spring”}\ \wedge\ year=2018}(section))$$

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

- To retrieve the social security numbers of all employees who either work in department 5 (RESULT1 below) or directly supervise an employee who works in department 5 (RESULT2 below).
- We can use the UNION operation as follows:

$$DEP5\_EMPS \leftarrow \sigma_{DNO=5}(EMPLOYEE)$$
$$RESULT1 \leftarrow \pi_{SSN}(DEP5\_EMPS)$$
$$RESULT2(SSN) \leftarrow \pi_{SUPERSSN}(DEP5\_EMPS)$$
$$RESULT \leftarrow RESULT1 \cup RESULT2$$

INTERSECTION

The set-intersection operation allows us to find tuples that are in both the input relations.

Assume: r, s have the same arity attributes of r and s are compatible

- Example: Find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters.

Query: $\Pi_{course\_id}(\sigma_{semester="Fall" \wedge year=2017}(section)) \cap \Pi_{course\_id}(\sigma_{semester="Spring" \wedge year=2018}(section))$

Output:

course_id
CS-101

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A

Example : Graduates ∩ Managers

Graduates

Number	Surname	Age
7274	Robinson	37
7432	O'Malley	39
9824	Darkes	38

Managers

Number	Surname	Age
9297	O'Malley	56
7432	O'Malley	39
9824	Darkes	38

Output:

Graduates ∩ Managers

Number	Surname	Age
7432	O'Malley	39
9824	Darkes	38

SET DIFFERENCE

The set-difference operation allows us to find tuples that are in one relation but are not in another.

Set differences must be taken between compatible relations. r and s must have the same arity. attribute domains of r and s must be compatible.

Example: to find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester.

Query: $\Pi_{course\_id}(\sigma_{semester="Fall" \wedge year=2017}(section)) - \Pi_{course\_id}(\sigma_{semester="Spring" \wedge year=2018}(section))$

Output:

course_id
CS-347
PHY-101

course_id	*sec_id*	*semester*	*year*	*building*	*room_number*	*time_slot_id*
BIO-101	1	Summer	2017	Painter	514	B
BIO-301	1	Summer	2018	Painter	514	A
CS-101	1	Fall	2017	Packard	101	H
CS-101	1	Spring	2018	Packard	101	F
CS-190	1	Spring	2017	Taylor	3128	E
CS-190	2	Spring	2017	Taylor	3128	A
CS-315	1	Spring	2018	Watson	120	D
CS-319	1	Spring	2018	Watson	100	B
CS-319	2	Spring	2018	Taylor	3128	C
CS-347	1	Fall	2017	Taylor	3128	A
EE-181	1	Spring	2017	Taylor	3128	C
FIN-201	1	Spring	2018	Packard	101	B
HIS-351	1	Spring	2018	Painter	514	C
MU-199	1	Spring	2018	Packard	101	D
PHY-101	1	Fall	2017	Watson	100	A

Graduates

Number	Surname	Age
7274	Robinson	37
7432	O'Malley	39
9824	Darkes	38

Managers

Number	Surname	Age
9297	O'Malley	56
7432	O'Malley	39
9824	Darkes	38

Output:

Graduates - Managers

Number	Surname	Age
7274	Robinson	37

Differences

Example : STUDENT ∪ INSTRUCTOR

STUDENT

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

Output:

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

Example : STUDENT ∩ INSTRUCTOR.

STUDENT

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

Output:

Fn	Ln
Susan	Yao
Ramesh	Shah

Example : STUDENT – INSTRUCTOR.

STUDENT

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

Output:

Fn	Ln
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

Example : INSTRUCTOR – STUDENT.

STUDENT

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

Output:

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

Equivalent Queries

Find information about courses taught by instructors in the Physics department with salary greater than 90,000.

Query 1: $\sigma_{dept\_name="Physics" \wedge salary > 90{,}000}(instructor)$

Query 2 : $\sigma_{dept\_name="Physics"}(\sigma_{salary > 90.000}(instructor))$

Instructor {ID, name, dept name, salary}

teaches (ID, course id, sec id, semester, year)

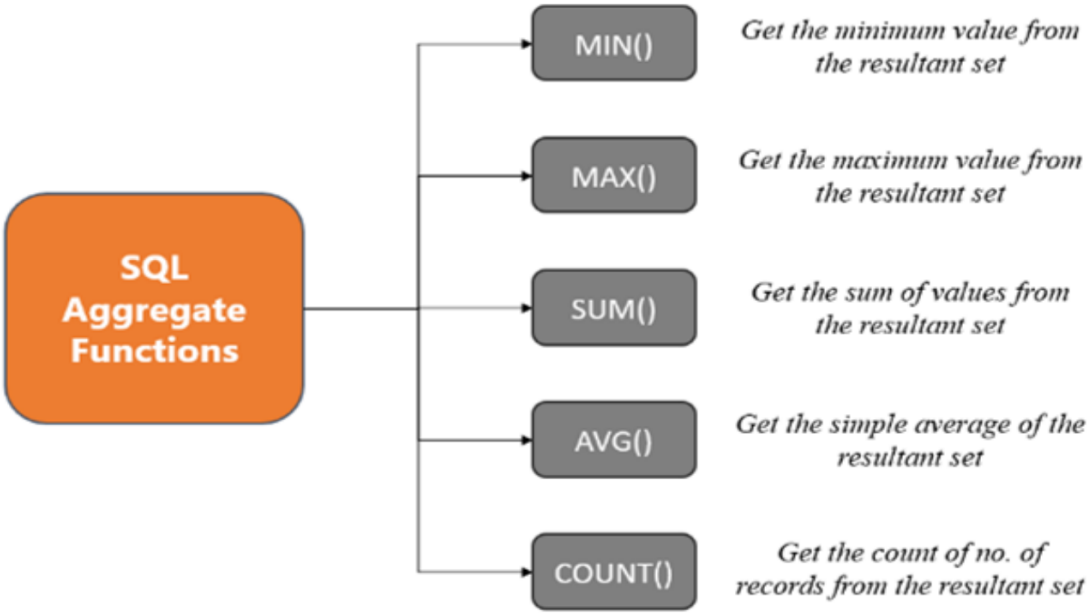
Example: Find information about courses taught by instructors in the Physics department.

Query 1: $\sigma_{dept\_name="Physics"}(instructor \bowtie_{instructor.ID = teaches.ID} teaches)$

Query 2: $(\sigma_{dept\_name="Physics"}(instructor)) \bowtie_{instructor.ID = teaches.ID} teaches$

Agregate Functions and Grouping

Aggregate functions are functions that take a collection (a set or multiset) of values as input and **return a single value.**



Common functions applied to collections of numeric values include SUM, AVERAGE, MAXIMUM, and MINIMUM

The COUNT function is used for counting tuples or values without removing duplicates

Average - returns the average of all non-Null values.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

π(DNO, COUNT(SSN), MAX(Salary), MIN(Salary), AVG(Salary)) (EMPLOYEE ⋈ DNO)

a. $\rho_{R(\text{Dno, No\_of\_employees, Average\_sal})}(_{\text{Dno}}\ \Im\ _{\text{COUNT Ssn, AVERAGE Salary}}\ (\text{EMPLOYEE}))$.

b. $_{\text{Dno}}\ \Im\ _{\text{Count ssn,Average\_salary}}(\text{EMPLOYEE})$.

c. $\Im\ _{\text{COUNT ssn, AVERAGE Salary}}(\text{EMPLOYEE})$.

Output:

R

(a)

Dno	No_of_employees	Average_sal
5	4	33250
4	3	31000
1	1	55000

(b)

Dno	Count_ssn	Average_salary
5	4	33250
4	3	31000
1	1	55000

(c)

Count_ssn	Average_salary
8	35125

Airplane mode off

Grouping

- In many cases we want to apply the aggregate functions to subgroups of tuples in a relation, where the subgroups are based on some attribute values.
- We can then apply the function to each such group independently to produce summary information about each group. SQL has a GROUP BY clause for this purpose.
- The GROUP BY clause specifies the grouping attributes, which should also appear in the SELECT
  clause, so that the value resulting from applying each aggregate function to a group of tuples appears along with the value of the grouping attribute(s)

Example: find the average salary of employees in each department or the number of employees who work on each project.

Query:

$$\rho_{R(\text{Dno,count(*), Avg(Salery)})}(\text{Dno } \Im \text{ (Dno,COUNT(*), AVG(Salary))} (\text{EMPLOYEE})$$

Output:

Fname	Minit	Lname	Ssn	...	Salary	Super_ssn	Dno
John	B	Smith	123456789		30000	333445555	5
Franklin	T	Wong	333445555		40000	888665555	5
Ramesh	K	Narayan	666884444		38000	333445555	5
Joyce	A	English	453453453	...	25000	333445555	5
Alicia	J	Zelaya	999887777		25000	987654321	4
Jennifer	S	Wallace	987654321		43000	888665555	4
Ahmad	V	Jabbar	987987987		25000	987654321	4
James	E	Bong	888665555		55000	NULL	1

Grouping EMPLOYEE tuples by the value of Dno

Dno	Count (*)	Avg (Salary)
5	4	33250
4	3	31000
1	1	55000

Result of Q24

Structured Query language (SQL)

SQL language : Considered one of the major reasons for the commercial success of relational databases

- Data Definition Language (DDL) statements are used to define the database structure or schema.
- Data Manipulation Language (DML) statements are used for managing data within schema objects
- DCL is the abstract of Data Control Language. Data Control Language includes commands such as GRANT, and is concerned with rights, permissions, and other controls of the database system.
- Transaction Control Language (TCL) is used to run the changes made by the DML statement.

Char	*Varchar*
Majorly use Char when you know that the strings are always going to be of the same size. Example : Pincode	Use Varchar when you have little idea about the size of the string that is going to be entered. Example :Name
Since it is a datatype of fixed length the storage taken for a Char will be equal to the length that it is initialised with.	Since it is a datatype of variable length the storage taken for a Char will be equal to the size of the string/data entered and not the entire length that is initialised with.
Char utilises the entire storage by the use of training spaces.	Varchar does not utilize the entire storage but only uses the storage based on the data input
Char gives better performance compared to Varchar	Varchar gives lower performance compared to Char
Char is incapable of holding a null value hence sql will automatically assign a Varchar for a null value	A Char null column is nothing but in reality a Varchar null column

Blob

- The full form of Blob is a Binary Large Object.
- This is used to store large binary data.
- This stores values in the form of binary streams.
- Using this you can stores files like videos, images, gifs, and audio files.
- MySQL supports this with the following datatypes:TINYBLOB
- BLOB
- MEDIUMBLOB
- LONGBLOB

Clob

- The full form of Clob is Character Large Object.
- This is used to store large textual data.
- This stores values in the form of character streams.
- Using this you can store files like text files PDF documents, word documents etc.
- MySQL supports this with the following datatypes:TINYTEXT
- TEXT
- MEDIUMTEXT
- LONGTEXT

Data Definition Language (DDL)

- CREATE - to create objects in the database
- ALTER - alters the structure of the database
- DROP - delete objects from the database
- TRUNCATE - remove all records from a table, including all spaces allocated for the records are removed
- COMMENT - add comments to the data dictionary
- RENAME - rename an object

DESCRIBE

Create Table from Existing Table:

- **Using `AS SELECT * FROM existing_table`:**

sql Copy code

```
CREATE TABLE new_table_name AS SELECT * FROM existing_table;
```

This form creates a new table (`new_table_name`) with the same structure and data as an existing table (`existing_table`).

- **Using `LIKE existing_table`:**

sql Copy code

```
CREATE TABLE new_table_name LIKE existing_table;
```

This form creates an empty table (`new_table_name`) with the same structure (columns and data types) as an existing table (`existing_table`), but it does not copy the data.

Alter Table Options

1) ADD

Example:

```
ALTER TABLE employees
ADD COLUMN birth_date DATE,
```

```
ADD COLUMN address VARCHAR(255);
```

Explanation: This adds two new columns (`birth_date` and `address`) to the existing `employees` table.

2) MODIFY

Example:

```
ALTER TABLE employees
MODIFY COLUMN salary DECIMAL(10,2),
MODIFY COLUMN department_id INT;
```

Explanation: This modifies the data type of the `salary` column and changes the data type of the `department_id` column.

3) DROP

Example:

```
ALTER TABLE employees
DROP COLUMN phone_number;
```

Explanation: This removes the `phone_number` column from the `employees` table.

4) RENAME

Example:

```
ALTER TABLE employees
RENAME COLUMN hire_date TO joining_date,
RENAME COLUMN job_title TO position;
```

Explanation: This renames the columns `hire_date` to `joining_date` and `job_title` to `position` in the `employees` table.

5) DESCRIBE TABLE

Example:

```
DESCRIBE employees;
```

Explanation: This statement provides information about the structure of the `employees` table, including column names, data types, and constraints.

6) SHOW CREATE TABLE

Example:

```
SHOW CREATE TABLE employees;
```

Explanation: This statement displays the SQL statement that was used to create the `employees` table, including constraints and indexes.

7) DROP TABLE

Example:

```
DROP TABLE employees;
```

Explanation: This removes the entire `employees` table along with its structure and data.

8) RENAME (Table)

Example:

```
RENAME TABLE employees TO staff;
```

Explanation: This renames the `employees` table to `staff`.

9) TRUNCATE

Example:

```
TRUNCATE TABLE employees;
```

Explanation: This deletes all rows from the `employees` table, but the table structure remains intact.

Constraints in SQL

Constraints in SQL refer to the conditions and restrictions that are applied on the database

If the condition we have applied to the database holds true for the data that is to be inserted, then only the data will be inserted into the database tables

- Key constraint: A primary key value cannot be duplicated
- Entity Integrity Constraint: A primary key value cannot be null
- Referential integrity constraints : The "foreign key " must have a value that is already present as a primary key, or may be null

Constraint	Description
NOT NULL	This constraint confirms that a column cannot store NULL value.
UNIQUE	This constraint ensures that each row for a column must have a different value.
PRIMARY KEY	This constraint is a combination of a NOT NULL constraint and a UNIQUE constraint. This constraint ensures that the specific column or combination of two or more columns for a table have a unique identity which helps to find a particular record in a table more easily and quickly.
CHECK	A check constraint ensures that the value stored in a column meets a specific condition.
DEFAULT	This constraint provides a default value when specified none for this column.
FOREIGN KEY	A foreign key constraint is used to ensure the referential integrity of the data. in one table to match values in another table.

- **SET NULL:** If a referenced record is deleted or updated, the foreign key values in the related records are set to NULL.

- **CASCADE:** If a referenced record is deleted or updated, the changes are propagated to the related records. For example, if a referenced record is deleted, all related records in other tables are also deleted.
- **SET DEFAULT:** Similar to SET NULL, but it sets the foreign key values to their default values instead.

DROP

One can also drop a whole schema if it is no longer needed by using the DROP SCHEMA command. There are two drop behavior options: CASCADE and RESTRICT.

DROP SCHEMA COMPANY CASCADE;
This removes the schema and all its elements including tables,views, constraints, etc

can get rid of the DEPENDENT relation by issuing the following command:
DROP TABLE DEPENDENT CASCADE;

If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is not referenced in any constraints

CREATE DATABASE;

USE DATBASE:

SHOW DATBASES:

DESC TABLE;

DROP DATABASE;

1. **ADD a column in the table:**

```
ALTER TABLE table_name
ADD new_column_name column_definition [FIRST | AFTER colum
n_name];
```

Example:

```
ALTER TABLE employees
ADD email VARCHAR(50) AFTER last_name;
```

2. **Add multiple columns in the table:**

```
ALTER TABLE table_name
ADD new_column_name1 column_definition [FIRST | AFTER colu
mn_name],
ADD new_column_name2 column_definition [FIRST | AFTER colu
mn_name];
```

Example:

```
ALTER TABLE employees
ADD email VARCHAR(50) AFTER last_name,
ADD phone VARCHAR(15) AFTER email;
```

3. **MODIFY column in the table:**

```
ALTER TABLE table_name
MODIFY column_name column_definition [FIRST | AFTER column
_name];
```

Example:

```
ALTER TABLE employees
MODIFY email VARCHAR(100) AFTER last_name;
```

4. **DROP column in the table:**

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

Example:

```
ALTER TABLE employees
DROP COLUMN phone;
```

5. **RENAME column in table:**

```
ALTER TABLE table_name
CHANGE COLUMN old_name new_name column_definition [FIRST |
AFTER column_name];
```

Example:

```
ALTER TABLE employees
CHANGE COLUMN email work_email VARCHAR(100) AFTER last_nam
e;
```

6. **RENAME table:**

```
ALTER TABLE table_name
RENAME TO new_table_name;
```

Example:

```
ALTER TABLE employees
RENAME TO staff;
```

7. **Altering (Changing) a Column Definition or a Name:**

```
-- Change data type of a column
ALTER TABLE table_name
MODIFY column_name new_data_type;

-- Change data type, make it NOT NULL, and set a default v
alue
```

```
ALTER TABLE table_name
MODIFY column_name new_data_type NOT NULL DEFAULT 100;

-- Change the default value of a column
ALTER TABLE table_name
ALTER COLUMN column_name SET DEFAULT 1000;
```

S.No	DROP	TRUNCATE
1.	It is used to eliminate the whole database from the table.	It is used to eliminate the tuples from the table.
2.	Integrity constraints get removed in the DROP command.	Integrity constraint doesn't get removed in the Truncate command.
3.	The structure of the table does not exist.	The structure of the table exists.
4.	Here the table is free from memory.	Here, the table is not free from memory.
5.	It is slow as compared to the TRUNCATE command.	It is fast as compared to the DROP command.

Database View and Priveleges

A database view is a virtual table that is based on the result of a SELECT query. It does not store the data itself but provides a way to represent the result of a query as if it were a table. Views can be used to simplify complex queries, encapsulate complex logic, and provide a security mechanism by restricting access to certain columns or rows.

```
CREATE VIEW view_name AS
SELECT column1, column2, ...
```

```
FROM table_name
WHERE condition;
```

```
Creating a simple view
CREATE VIEW employee_view AS
SELECT emp_id, emp_name, salary
FROM employee
WHERE department = 'IT';
```

Example:

Create a view that lists all course sections offered by the Physics department in the Fall 2017 semester with the building and room number of each section.

Query:

```
create view physics_fall_2017 as
select course.course_id, sec_id, building, room_number
from course, section
where course.course_id = section.course_id
and course.dept_name = 'Physics'
and section.semester = 'Fall';
```

Output:

course_id	sec_id	building	room_number
PHY-101	1	Watson	100

Example:

Retrieve all Physics courses offered in the Fall 2017 semester in the Watson building.

Query:

```
select course id
from physics_fall_2017
where building = 'Watson';
```

Output:

course_id
PHY-101

Retrieve the course ID and room number of all Physics courses offered in the Fall 2017 semester in the Watson building.

Query:

```
create view physics_fall_2017_watson as
select course_id, room_number
from physics_fall_2017
where building = 'Watson';
```

Output:

course_id	room_number
PHY-101	100

View Advantages

- Complexity: Views help to reduce complexity. Different views can be created on the same
  base table for different users.
- Security: It increases security by excluding sensitive information from the view.
- Query Simplicity: It helps to simplify commands from the user. A view can draw data from
  several different tables and present it as a single table.

- Consistency: A view can present a consistent, unchanged image of the structure of the
  database. Views can be used to rename columns without affecting the base table.
- Data Integrity: If data is accessed and entered through a view, the DBMS can
  automatically check the data to ensure that it meets the specified integrity constraints.
- Storage Capacity: Views take very little space to store the data.
- Logical Data Independence: Views can make the application and database tables to a
  certain extent independent.

Disadvantages

- You cannot INSERT if the base table has any not null columns that do not appear in the view.
- You cannot INSERT or UPDATE if any of the columns referenced in the INSERT or UPDATE
  contains group functions or columns defined by an expression.
- You can't execute INSERT, UPDATE, DELETE statements on a view if with READ ONLY option is enabled.
- You can't create a view using temporary tables.
- You cannot INSERT, UPDATE, or DELETE if the view contains group functions GROUP BY,
  DISTINCT, or a reference to a pseudo column row_num.
- You can't pass parameters to the SQL server views.
- You can't associate rules and defaults with views.

Priveleges

Creating a new user

```
CREATE USER ‘aryan’@’localhost’
IDENTIFIED BY ‘mypassword’;
```

To grant privileges to a user

grant <privilege list>
on <table name or view name>
to <user/role list>;

1. **Granting Privileges:**
   The SQL standard provides the `GRANT` statement to assign specific privileges to a user or role on a table or view.

   ```
   -- Granting SELECT and INSERT privileges on a table
   GRANT SELECT, INSERT ON table_name TO user_name;
   ```

   Example:

   ```
   -- Granting SELECT privilege on all tables within Unive
   rsity_db to Charles
   GRANT SELECT ON University_db.* TO Charles;
   ```

2. **Checking Privileges:**
   To check the privileges assigned to a specific user, you can use the `SHOW GRANTS` command.

   ```
   -- Show privileges for a specific user
   SHOW GRANTS FOR user_name;
   ```

   Example:

```
-- Show privileges for Charles
SHOW GRANTS FOR Charles;
```

3. **Revoking Privileges:**
   To revoke an authorization, the `REVOKE` statement is used. It has a similar syntax to `GRANT`.

   ```
   -- Revoking SELECT privilege on a table
   REVOKE SELECT ON table_name FROM user_name;
   ```

   Example:

   ```
   -- Revoking SELECT privilege on all tables within Unive
   rsity_db from Charles
   REVOKE SELECT ON University_db.* FROM Charles;
   ```

These statements allow database administrators to control and manage user access rights to tables and views within a database.

Database Modification

SELECT

The SELECT clause lists the attributes desired in the result of a query.

It corresponds to the projection operation of relational algebra.

FROM clause is used to specify the table name or the relation name

Example: Find the names of all instructors:

```
select fname,Lname
from EMPLOYEE;
```

Output:

Fname	Lname
John	Smith
John	Smith
Franklin	Wong
Joyce	English
Ramesh	Narayan
James	Borg

INSERT

- In its simplest form, INSERT is used to add a single tuple (row) to a relation (table)
- We must specify the relation name and a list of values for the tuple.

Form 1:

The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE command.

Syntax:

INSERT INTO table_name VALUES (value1,value2,…..);

Form 2

Syntax:

INSERT INTO table_name(attribute1,attribute2,….) VALUES (value1,value2,…..);

- Form 2 allows the user to specify explicit attribute names that correspond to the
  values provided in the INSERT command.
- This is useful if a relation has many attributes but only a few of those attributes are assigned values in the new tuple.
- However, the values must include all attributes with NOT NULL specification and no default value

```
INSERT INTO WORKS_ON_INFO (Emp_name, Proj_name, Hours_per_
week)
SELECT E.Lname, P.Pname, W.Hours
FROM PROJECT P, WORKS_ON W, EMPLOYEE E
WHERE P.Pnumber = W.Pno AND W.Essn = E.Ssn;
```

- This query inserts data into the `WORKS_ON_INFO` table.
- The data is selected from the result of a join between the `PROJECT`, `WORKS_ON`, and `EMPLOYEE` tables.
- It selects the last name of employees (`E.Lname`), project names (`P.Pname`), and hours per week (`W.Hours`) where there are matches in project numbers and employee social security numbers.

```
INSERT INTO WORKS_ON_INFO_Backup (SELECT * FROM WORKS_ON_I
NFO);
```

- This query creates a backup of the data in the `WORKS_ON_INFO` table.
- It inserts all columns and rows from `WORKS_ON_INFO` into a table named `WORKS_ON_INFO_Backup`.
- The condition mentioned is that the schema (column structure) of both tables (`WORKS_ON_INFO` and `WORKS_ON_INFO_Backup`) must be the same.

UPDATE:

The `UPDATE` command is used to modify attribute values of one or more selected tuples in a relational database. It is particularly useful for making changes to existing data based on specified conditions.

Syntax:

```
UPDATE table_name SET column_name = value WHERE condition;
```

Examples:

1. *Changing Project Information:*

   ```
   -- Change the location and controlling department numbe
   r of project number 10
   UPDATE PROJECT SET Plocation = 'Bellaire', Dnum = 5 WHE
   RE Pnumber = 10;
   ```

   Before:

Pnumber	Plocation	Dnum
10	Houston	1

   After:

Pnumber	Plocation	Dnum
10	Bellaire	5

2. *Salary Raise for Research Department:*

   ```
   -- Give all employees in the 'Research' department a 1
   0% raise in salary
   UPDATE EMPLOYEE SET Salary = Salary * 1.1 WHERE Dno =
   5;
   ```

   Before:

Emp_id	Salary	Dno

101	60000	5
102	70000	5

After:

Emp_id	Salary	Dno
101	66000	5
102	77000	5

3. *Salary Raise for Below-Average Employees:*

```
-- Give a 5 percent salary raise to employees whose sal
ary is less than average
UPDATE EMPLOYEE SET Salary = Salary * 1.05 WHERE Salary
< (SELECT AVG(Salary) FROM EMPLOYEE);
```

Before:

Emp_id	Salary	Dno
201	55000	3
202	60000	3

After:

Emp_id	Salary	Dno
201	57750	3
202	63000	3

DELETE Command:

The `DELETE` command is used to remove tuples from a relation based on specified conditions. It is useful for eliminating unwanted records from a table.

Syntax:

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

Examples:

1. *Delete Employee Record by Last Name:*

```
-- Delete the employee record whose last name is 'Zelay
a'
DELETE FROM EMPLOYEE WHERE Lname = 'Zelaya';
```

Before:

Emp_id	Lname	Salary
301	Zelaya	75000
302	Smith	80000

After:

Emp_id	Lname	Salary
302	Smith	80000

2. *Delete All Employee Records:*

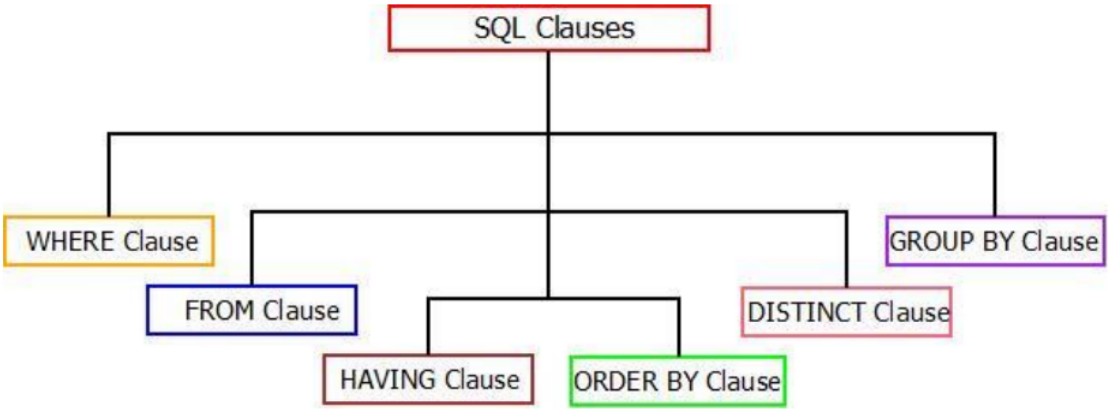
```
-- Delete all employee records (similar to truncating t
he employee table)
DELETE FROM EMPLOYEE;
```

Before:

Emp_id	Lname	Salary
401	Brown	90000
402	White	95000

After:
(The table becomes empty)

SQL Clauses:




```
SELECT [ALL | DISTINCT | DISTINCTROW]
[HIGH_PRIORITY]
[FROM table_references [PARTITION partition_list]]
[WHERE where_condition]
[GROUP BY {col_name | expr | position}]
[HAVING where_condition]
[ORDER BY {col_name | expr | position} [ASC | DESC]]
[LIMIT {[offset,] row_count | row_count OFFSET offset}];
```

1. ALL | DISTINCT | DISTINCT ROW:

Example:

```
-- Before: EMPLOYEE table
-- SSN | Fname | Lname  | Gender
-- ----|-------|--------|--------
-- 1   | John  | Doe    | Male
-- 2   | Jane  | Smith  | Female
-- 3   | Bob   | Johnson| Male

-- After: DISTINCT employee names
SELECT DISTINCT Fname, Lname FROM EMPLOYEE;
```

```
-- Fname | Lname
-- ------|-------
-- John  | Doe
-- Jane  | Smith
-- Bob   | Johnson
```

Description: The `DISTINCT` keyword is used to retrieve unique combinations of the specified columns from the result set.

2. HIGH_PRIORITY:

Example:

```
-- Before: EMPLOYEE table
-- SSN | Fname | Lname  | Gender
-- ----|-------|--------|--------
-- 1   | John  | Doe    | Male
-- 2   | Jane  | Smith  | Female
-- 3   | Bob   | Johnson| Male

-- After: HIGH_PRIORITY employee names
SELECT HIGH_PRIORITY Fname, Lname FROM EMPLOYEE;
-- Fname | Lname
-- ------|-------
-- John  | Doe
-- Jane  | Smith
-- Bob   | Johnson
```

Description: The `HIGH_PRIORITY` keyword suggests that the query should be given higher priority, potentially affecting the order in which it is executed.

3. FROM table_references:

Example:

```
-- Before: EMPLOYEE table
-- SSN | Fname | Lname  | Gender
-- ----|-------|--------|--------
-- 1   | John  | Doe    | Male
-- 2   | Jane  | Smith  | Female
-- 3   | Bob   | Johnson| Male

-- After: Employee names from EMPLOYEE table
SELECT Fname, Lname FROM EMPLOYEE;
-- Fname | Lname
-- ------|-------
-- John  | Doe
-- Jane  | Smith
-- Bob   | Johnson
```

Description: The `FROM` clause specifies the table from which the data is selected.

4. WHERE where_condition:

Example:

```
-- Before: EMPLOYEE table
-- SSN | Fname | Lname  | Gender | Dno | Salary
-- ----|-------|--------|--------|-----|-------
-- 1   | John  | Doe    | Male   | 5   | 65000
-- 2   | Jane  | Smith  | Female | 5   | 72000
-- 3   | Bob   | Johnson| Male   | 7   | 60000

-- After: Employees in department 5 with salary < $70,000
SELECT Fname, Salary FROM EMPLOYEE WHERE Dno = 5 AND Salar
y < 70000;
-- Fname | Salary
-- ------|-------
-- John  | 65000
```

Description: The `WHERE` clause filters the rows based on a specified condition.

5. GROUP BY {col_name | expr | position}:

Example:

```
-- Before: EMPLOYEE table
-- SSN | Fname | Lname  | Gender | Dno | Salary
-- ----|-------|--------|--------|-----|-------
-- 1   | John  | Doe    | Male   | 5   | 65000
-- 2   | Jane  | Smith  | Female | 5   | 72000
-- 3   | Bob   | Johnson| Male   | 7   | 60000

-- After: Count of employees by gender
SELECT COUNT(SSN), Gender FROM EMPLOYEE GROUP BY Gender;
-- COUNT(SSN) | Gender
-- -----------|--------
-- 2          | Male
-- 1          | Female
```

Description: The `GROUP BY` clause groups the result set by specified columns and allows the use of aggregate functions.

6. HAVING where_condition:

Example:

```
-- Before: EMPLOYEE table
-- SSN | Fname | Lname  | Gender | Dno | Salary
-- ----|-------|--------|--------|-----|-------
-- 1   | John  | Doe    | Male   | 5   | 65000
-- 2   | Jane  | Smith  | Female | 5   | 72000
-- 3   | Bob   | Johnson| Male   | 7   | 60000

-- After: Departments with more than 1 employee
SELECT COUNT(SSN), Dno FROM EMPLOYEE GROUP BY Dno HAVING C
```

```
OUNT(SSN) > 1;
-- COUNT(SSN) | Dno
-- -----------|-----
-- 2          | 5
```

Description: The `HAVING` clause filters the result set based on aggregate function results.

7. ORDER BY {col_name | expr | position} [ASC | DESC]:

Example:

```
-- Before: EMPLOYEE table
-- SSN | Fname | Lname   | Gender | Dno | Salary
-- ----|-------|--------|--------|-----|-------
-- 1   | John  | Doe     | Male   | 5   | 65000
-- 2   | Jane  | Smith   | Female | 5   | 72000
-- 3   | Bob   | Johnson| Male   | 7   | 60000

-- After: Employees by descending order of salary
SELECT Fname, Lname, Salary FROM EMPLOYEE ORDER BY Salary
DESC;
-- Fname | Lname    | Salary
-- ------|---------|-------
-- Jane  | Smith    | 72000
-- John  | Doe      | 65000
-- Bob   | Johnson | 60000
```

Description: The `ORDER BY` clause sorts the result set by specified columns in ascending (`ASC`) or descending (`DESC`) order.

8. LIMIT {[offset,] row_count | row_count OFFSET offset}:

Example:

```
-- Before: EMPLOYEE table
-- SSN | Fname | Lname   | Gender | Dno | Salary
-- ----|-------|--------|--------|-----|-------
-- 1   | John  | Doe     | Male   | 5   | 65000
-- 2   | Jane  | Smith   | Female | 5   | 72000
-- 3   | Bob   | Johnson| Male   | 7   | 60000

-- After: Limited to 2 rows
SELECT Fname, Lname FROM EMPLOYEE LIMIT 2;
-- Fname | Lname
-- ------|-------
-- John  | Doe
-- Jane  | Smith
```

Description: The `LIMIT` clause restricts the number of rows returned in the result set.

Examples: Find all employees who were born during the 1960s.

Query: SELECT Fname, Lname, bdate FROM EMPLOYEE WHERE Bdate LIKE '__6_____';

Output:

Fname	Lname	bdate
John	Smith	1965-01-09
Ramesh	Narayan	1962-09-15
Ahmed	Jabbar	1969-03-29
Alicia	Zelaya	1968-01-19

- SQL includes a string-matching operator for comparisons on character strings.
- The operator **LIKE** uses patterns that are described using two special characters:

  percent (%): The % character matches any substring.

  underscore (_): The _ character matches any character.

Example: Retrieve all employees whose address is in Houston, Texas.

Query: SELECT Fname, Lname FROM EMPLOYEE WHERE Address LIKE '%Houston,TX%';

Output:

Fname	Lname	Address
John	Smith	731 Fondren,Houston,TX
Franklin	Wong	638 voss,Houston,TX
Joyce	English	5631 Rice,Houston,TX
James	Borg	450 Stone, Houston,TX
Ahmed	Jabbar	980 Dallas, Houston,TX

SET OPERATIONS

Union

- The SQL UNION operation is used to combine the results of two or more SQL SELECT queries.
- In the UNION operation, the number of columns and datatypes must be the same in both the tables on which the UNION operation is being applied.
- It eliminates duplicate rows from its result set.

name	course	grade
Alex	Physics	S
Bert	Physics	S
Charles	Physics	A
Dennis	Physics	B
Evans	Physics	B

name	course	grade
Alex	Mathematics	S
Bert	Mathematics	S
Charles	Mathematics	A
Dennis	Mathematics	A
Evans	Mathematics	A

Query :

```
SELECT grade FROM student_course
WHERE course = 'Physics'
UNION
SELECT grade FROM student_course
WHERE course = 'Mathematics';
```

Output :

grade
S
A
B

name	course	grade
Alex	Physics	S
Bert	Physics	S
Charles	Physics	A
Dennis	Physics	B
Evans	Physics	B

name	course	grade
Alex	Mathematics	S
Bert	Mathematics	S
Charles	Mathematics	A
Dennis	Mathematics	A
Evans	Mathematics	A

Union All

The UNION ALL operation is similar to the UNION operation. It returns the set without removing duplicates and sorting the data.

Query:

```
SELECT grade FROM student_course
WHERE course = 'Physics'
UNION ALL
SELECT grade FROM student_course
WHERE course = 'Mathematics';
```

Output :

grade
S
S
A
B
B
S
S
A
A
A

name	course	grade
Alex	Physics	S
Bert	Physics	S
Charles	Physics	A
Dennis	Physics	B
Evans	Physics	B

name	course	grade
Alex	Mathematics	S
Bert	Mathematics	S
Charles	Mathematics	A
Dennis	Mathematics	A
Evans	Mathematics	A

Query:

```
SELECT distinct grade FROM student_course
WHERE course = 'Physics'
UNION ALL
SELECT distinct grade FROM student_course
WHERE course = 'Mathematics';
```

Output :

grade
S
A
B
S
A

name	course	grade
Alex	Physics	S
Bert	Physics	S
Charles	Physics	A
Dennis	Physics	B
Evans	Physics	B

name	course	grade
Alex	Mathematics	S
Bert	Mathematics	S
Charles	Mathematics	A
Dennis	Mathematics	A
Evans	Mathematics	A

Notice the difference between the results of the two UNION ALL operations. This is due to the inclusion of the DISTINCT keyword. DISTINCT eliminates any duplicates and hence, the output of each SELECT statement has no duplicates. However, due to the usage of UNION ALL, duplicates are not eliminated in the result of the UNION ALL operation.

Intersect

- It is used to combine two SELECT statements. The INTERSECT operation returns common rows from both the SELECT statements.
- In the Intersect operation, the number of columns and corresponding datatypes must be the same.
- It has no duplicates

Query :

```
SELECT grade FROM student_course
WHERE course = 'Physics'
INTERSECT
SELECT grade FROM student_course
WHERE course = 'Mathematics';
```

Output :

grade
S
A

name	course	grade
Alex	Physics	S
Bert	Physics	S
Charles	Physics	A
Dennis	Physics	B
Evans	Physics	B

name	course	grade
Alex	Mathematics	S
Bert	Mathematics	S
Charles	Mathematics	A
Dennis	Mathematics	A
Evans	Mathematics	A

Intersect All

INTERSECT ALL is similar to the INTERSECT operation. It helps retain duplicates.

Query:

```
SELECT grade FROM student_course
WHERE course = 'Physics'
INTERSECT ALL
SELECT grade FROM student_course
WHERE course = 'Mathematics';
```

Output :

grade
S
S
A

name	course	grade
Alex	Physics	S
Bert	Physics	S
Charles	Physics	A
Dennis	Physics	B
Evans	Physics	B

name	course	grade
Alex	Mathematics	S
Bert	Mathematics	S
Charles	Mathematics	A
Dennis	Mathematics	A
Evans	Mathematics	A

Except

- It combines the result of two SELECT statements.
- EXCEPT operator is used to display the rows that are present in the first query but absent in the second query.
- It has no duplicates

c1-c2

Query :

```
SELECT grade FROM student_course
WHERE course = 'Physics'
EXCEPT
SELECT grade FROM student_course
WHERE course = 'Mathematics';
```

```
SELECT grade FROM student_course
WHERE course = 'Mathematics'
EXCEPT
SELECT grade FROM student_course
WHERE course = 'Physics';
```

Output :

grade
B

grade

(Empty Set)

name	course	grade
Alex	Physics	S
Bert	Physics	S
Charles	Physics	A
Dennis	Physics	B
Evans	Physics	B

name	course	grade
Alex	Mathematics	S
Bert	Mathematics	S
Charles	Mathematics	A
Dennis	Mathematics	A
Evans	Mathematics	A

We get unique elements

Except All

EXCEPT ALL is similar to the EXCEPT operation. It helps retain duplicates

Query :

```
SELECT grade FROM student_course
WHERE course = 'Physics'
EXCEPT ALL
SELECT grade FROM student_course
WHERE course = 'Mathematics';
```

Output :

grade
B
B

name	course	grade
Alex	Physics	S
Bert	Physics	S
Charles	Physics	A
Dennis	Physics	B
Evans	Physics	B

name	course	grade
Alex	Mathematics	S
Bert	Mathematics	S
Charles	Mathematics	A
Dennis	Mathematics	A
Evans	Mathematics	A

```
SELECT grade FROM student_course
WHERE course = 'Mathematics'
EXCEPT ALL
SELECT grade FROM student_course
WHERE course = 'Physics';
```

grade
A
A

NULL VALUES

1. **Unknown value**: value exists but is not known, or it is not known whether or not the
   value exists. For example, when a person's date of birth is not known, it is represented
   as NULL in the database.
2. **Unavailable or withheld value**: value exists but is purposely withheld. For example, a
   person might not want to disclose his/her date of birth.
3. **Not applicable attributes:** the attribute does not apply to this tuple or is undefined for
   this tuple. For example, an attribute LastCollegeDegree would be NULL for a person
   who has no college degrees.

In other words, any comparison operation involving NULL values would result in an,UNKNOWN.

(a)	**AND**	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	FALSE	UNKNOWN
	FALSE	FALSE	FALSE	FALSE
	UNKNOWN	UNKNOWN	FALSE	UNKNOWN
(b)	**OR**	TRUE	FALSE	UNKNOWN
	TRUE	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	UNKNOWN
	UNKNOWN	TRUE	UNKNOWN	UNKNOWN
(c)	**NOT**			
	TRUE	FALSE		
	FALSE	TRUE		
	UNKNOWN	UNKNOWN		

False AND unknown = False

True OR unknown = True

NOT unknown = unknown

1. If the WHERE predicate evaluates to FALSE or UNKNOWN for a tuple, the tuple is not added to the result.

Imagine you have a list of employees, and you want to find those who work more than 40 hours a week. If an employee's working hours are not known (NULL) or less than 40, they won't be included in the result.

Example:

```
-- Before: WORKS_ON table
-- Essn | Hours
-- -----|------
-- 1    | 45
```

```
-- 2    | NULL
-- 3    | 35

-- After: Finding employees working more than 40 hours
SELECT Essn, Hours FROM WORKS_ON WHERE Hours > 40;
```

2. When a JOIN condition is specified, tuples with NULL values for the join attributes are not included in the result (unless it is an OUTER JOIN)

Let's say you have two tables - one with employee information and another with department information. If you're joining them based on a department number, and some employees don't have a department assigned (NULL), they won't be included in the result.

Example:

```
-- Before: EMPLOYEE table
-- Ssn | DeptNo
-- ----|-------
-- 1   | 101
-- 2   | NULL
-- 3   | 102

-- Before: DEPARTMENT table
-- DeptNo | DeptName
-- -------|---------
-- 101    | HR
-- 102    | IT

-- After: Joining tables on DeptNo
SELECT E.Ssn, E.DeptNo, D.DeptName
FROM EMPLOYEE E
JOIN DEPARTMENT D ON E.DeptNo = D.DeptNo;
```

3. SQL uses comparison operators IS and IS NOT to check if an attribute value is NULL or not, respectively

In simple terms, you can use `IS NULL` to check if a value is NULL and `IS NOT NULL` to check if a value is not NULL.

Example:

```
-- Before: EMPLOYEE table
-- Ssn | DeptNo
-- ----|-------
-- 1   | NULL
-- 2   | 102
-- 3   | 103

-- After: Selecting employees without a department
SELECT Ssn FROM EMPLOYEE WHERE DeptNo IS NULL;
```

4. SQL allows us to test whether the result of a comparison is UNKNOWN, rather than TRUE or FALSE, by using the clauses IS UNKNOWN and IS NOT UNKNOWN.

This is a way to handle cases where a comparison involving NULL values results in an unknown outcome.

Example:

```
-- Before: EMPLOYEE table
-- Ssn | Salary
-- ----|-------
-- 1   | NULL
-- 2   | 50000
-- 3   | 60000

-- After: Checking if salary is unknown
SELECT Ssn FROM EMPLOYEE WHERE Salary = 50000 IS UNKNOWN;
```

5. When a query uses the SELECT DISTINCT clause, duplicate tuples are eliminated.

Using `DISTINCT` ensures that you only get unique rows in your result, treating NULL values as identical.

Example:

```
-- Before: EMPLOYEE table
-- Ssn | DeptNo
-- ----|-------
-- 1   | 101
-- 2   | 102
-- 3   | 101
-- 4   | NULL

-- After: Selecting distinct departments
SELECT DISTINCT DeptNo FROM EMPLOYEE;
```

6. Treatment of NULL in DISTINCT is different from NULL values in predicates.

In the context of DISTINCT, if two tuples have the same values (including NULL) for all attributes, only one copy is retained.

Example:

```
-- Before: EMPLOYEE table
-- Fname | Lname
-- ------|------
-- John  | Doe
-- Jane  | NULL
-- John  | Doe

-- After: Selecting distinct names
SELECT DISTINCT Fname, Lname FROM EMPLOYEE;
```

7. The approach of treating tuples as identical if they have the same values for all attributes, even if some of the values are NULL, is also used for set operations UNION, INTERSECTION, and EXCEPT.

When combining or comparing sets, if two tuples have identical values for all attributes (including NULLs), they are treated as the same.

Example:

```
-- Before: EMPLOYEE table
-- Ssn | DeptNo
-- ----|-------
-- 1   | 101
-- 2   | NULL
-- 3   | 101
-- 4   | 102

-- After: Union of distinct departments
SELECT DeptNo FROM EMPLOYEE
UNION
SELECT DeptNo FROM DEPARTMENT;
```

Agregate Functions

Aggregate functions are used to summarize information from multiple tuples into a singletuple summary

- **COUNT** - returns the number of tuples or values specified in a query
- **SUM** – returns the sum of a set (or multiset) of numeric values
- **MAX** – returns the maximum value from a set(or multiset) of numeric values
- **MIN** - returns the minimum value from a set(or multiset) of numeric values
- **AVG** – returns the average of a set(or multiset) of numeric values

Example: **Find the sum of salaries of all employees, the maximum salary, the minimum salary, and the average salary**

Query: SELECT SUM(Salary), MAX(Salary), MIN(Salary), AVG(Salary) FROM EMPLOYEE;

Output:

SUM(Salary)	MAX(Salary)	MIN(Salary)	AVG(Salary)
281000.00	55000.00	25000.00	35125.000000

Example: **Retrieve the total number of employees in the company.**

Query: SELECT COUNT(*) FROM EMPLOYEE;

Output:

COUNT(*)
8

Example: **Count the number of distinct salary values in the database.**

Query: SELECT COUNT(DISTINCT Salary) FROM EMPLOYEE;

Output:

COUNT(DISTINCT Salary)
6

In general, when an aggregate function is applied to a collection of values, NULL values are discarded before the calculation.

If the collection becomes empty because all the values are NULL, COUNT returns zero and other aggregate functions return NUL

Example: **Find the average number of hours that employees work on each project.**

Query: SELECT Pno, AVG(Hours) as avg_hours FROM WORKS_ON GROUP BY Pno;

SQL has two functions that are applied to Boolean values – SOME and ALL.

- The SOME function returns True if at least one element in the collection is True.
- The ALL function returns True if all the elements in the collection are True.

Grouping

The attributes used to partition a relation into groups are called grouping attributes

If NULLs exist in the grouping attribute, then a separate group is created for all tuples
with a NULL value in the grouping attribute.

Example: **For each department, retrieve the department number, the number of employees in the department, and their average salary.**

Query: SELECT Dno, COUNT(*), AVG (Salary) FROM EMPLOYEE GROUP BY Dno;

Output:

Dno	COUNT(*)	AVG (Salary)
1	1	55000.000000
4	3	31000.000000
5	4	33250.000000

- In the above query, *Dno* is the grouping attribute. The *EMPLOYEE* relation is partitioned into groups in such a way that each group has tuples with the same value of *Dno*.
- Hence, each group consists of employees who work for the same department.
- The COUNT and AVG functions are applied to each group independently and are displayed in the result relation.
- The SELECT clause must contain only the grouping attributes and aggregate functions applied on each group of tuples.

Example: **For each project, retrieve the project number, the project name, and the number of employees who work on that project.**

Query: SELECT Pnumber, Pname, COUNT(*) FROM PROJECT, WORKS_ON
WHERE Pnumber = Pno GROUP BY Pnumber, Pname;

Output:

Pnumber	Pname	COUNT(*)
10	Computerization	3
30	Newbenefits	3
1	ProductX	2
2	ProductY	3
3	ProductZ	2
20	Reorganization	3

- The above query is an example of using GROUP BY along with JOIN.
- In such statements, relations are first joined using the condition specified in the WHERE clause. Grouping and aggregation are then performed on the result of this joining

The HAVING clause is used to specify conditions for groups of tuples, whereas the WHERE clause is used to specify conditions for individual tuples.

Example:

For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.

Query:

SELECT Pnumber, Pname, COUNT(*) FROM PROJECT, WORKS_ON
WHERE Pnumber = Pno GROUP BY Pnumber, Pname HAVING COUNT(*) > 2;

Output:

Pnumber	Pname	COUNT(*)
10	Computerization	3
30	Newbenefits	3
2	ProductY	3
20	Reorganization	3

Example: **For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.**

Query: SELECT Pnumber, Pname, COUNT (*)
FROM PROJECT, WORKS_ON, EMPLOYEE
WHERE Pnumber = Pno AND Ssn = Essn AND Dno = 5
GROUP BY Pnumber, Pname;

Output:

Pnumber	Pname	COUNT(*)
1	ProductX	2
2	ProductY	3
3	ProductZ	2
10	Computerization	1
20	Reorganization	1

The rule is that the condition specified in the WHERE clause is executed first, to select the individual tuples, and then the HAVING clause is executed, to select groups of tuples.

This means that the HAVING clause groups only those tuples that are selected by the WHERE clause.

Retrieve the count of number of employees whose salary exceeds $40,000 in each department, but only for those departments where more than two employees work.

The correct query is given below

Query: SELECT Dno, COUNT(*) FROM EMPLOYEE
WHERE Salary>40000 AND Dno IN
(SELECT Dno FROM EMPLOYEE GROUP BY Dno HAVING COUNT(*) > 2)
GROUP BY Dno;

Output:

Dno	COUNT(*)
4	1

Nested Queries

A nested query can be defined as a complete SELECT-FROM-WHERE block within another SQL query. The other query is called the outer query.

SQL provides a comparison operator IN, which checks if a value is present in a set(or multiset) of values. It returns True if the value is present in the set and False otherwise. It essentially checks for Set membership

Example: **Retrieve the list of project numbers of projects that have an employee with the last name 'Smith' involved either as a manager or as a worker.**

UNIVERSITY

Query:
```
SELECT DISTINCT Pnumber
    FROM PROJECT WHERE Pnumber IN
    ( SELECT Pnumber FROM PROJECT, DEPARTMENT, EMPLOYEE
    WHERE Dnum = Dnumber AND Mgr_ssn = Ssn AND Lname = 'Smith' )
    OR Pnumber IN
    ( SELECT Pno FROM WORKS_ON, EMPLOYEE
    WHERE Essn = Ssn AND Lname = 'Smith' );
```

Output:

Pnumber
1
2

In general, a nested query returns a table(relation) which is a set(or multiset) of tuples.

However, when a nested query returns a single value, we can use the = operator instead of the IN operator.

Example: Retrieve the ssn of all employees who work on the same (project, hours) combination on some project that employee 'John Smith' (ssn = '123456789') works on.

UNIVERSITY

Query:
```
SELECT DISTINCT Essn FROM WORKS_ON
    WHERE (Pno, Hours) IN
    ( SELECT Pno, Hours FROM WORKS_ON
    WHERE Essn = '123456789' );
```

Output:

Essn
123456789

- comp_op denotes comparison operators such as =, <> (not equals), >, >=, <, <= which are
  used to compare two values.
- comp_op **ALL returns True if the result of applying comp_op between the single value and an element of the set is True for every element of the set.**
- comp_op **SOME(or ANY) returns True if the result of applying comp_op between the single value and an element of the set is True for at least one element in the set.**

Example: **Retrieve the last and first names of all employees whose salary is greater than the salary of all the employees in department 5**

Query: SELECT Lname, Fname FROM EMPLOYEE

WHERE Salary > ALL (SELECT Salary FROM EMPLOYEE WHERE Dno = 5);

Example: **Retrieve the last and first names of all employees whose salary is greater than the salary of at least one employee in department 5**

Query: SELECT Lname, Fname FROM EMPLOYEE

WHERE Salary > SOME (SELECT Salary FROM EMPLOYEE WHERE Dno = 5);

Using aliases makes the code more readable, especially in complex queries with multiple tables.

Correlated Nested queries

A correlated nested query can be defined as a nested query in which the WHERE clause references attributes of a relation declared in the outer query.

Example: Retrieve the name of each employee who has a dependent who is of the same sex as the employee.

Query: SELECT E.Fname, E.Lname FROM EMPLOYEE AS E WHERE E.Ssn IN

(SELECT D.Essn FROM DEPENDENT AS D WHERE E.Gender = D.Gender);

Output:

Fname	Lname
Franklin	Wong

EXISTS and UNIQUE are Boolean functions that can be used in the WHERE clause

The EXISTS function in SQL is used to check whether the result of a nested query is
empty (contains no tuples) or not.

Example: Retrieve the name of each employee who has a dependent who is of the same sex as the employee.

Query: SELECT E.Fname, E.Lname FROM EMPLOYEE AS E
WHERE EXISTS (SELECT * FROM DEPENDENT AS D
WHERE E.Ssn = D.Essn AND E.Gender = D.Gender);

Output:

Fname	Lname
Franklin	Wong

Example: Retrieve the names of employees who have no dependents

Query: SELECT Fname, Lname FROM EMPLOYEE
WHERE NOT EXISTS (SELECT * FROM DEPENDENT WHERE Ssn = Essn);

Output:

Fname	Lname
Joyce	English
Ramesh	Narayan
James	Borg
Ahmed	Jabbar
Alicia	Zelaya

Example: List the names of managers who have at least one dependent.

Query: SELECT Fname, Lname FROM EMPLOYEE
WHERE EXISTS (SELECT * FROM DEPENDENT WHERE Ssn = Essn)
AND EXISTS (SELECT * FROM DEPARTMENT WHERE Ssn = Mgr_ssn);

Output:

Fname	Lname
Franklin	Wong
Jennifer	Wallace

Example: Find the average employee salary of those departments where the average salary is greater than $32,000.

```
Query: SELECT Dno, ROUND(AVG(Salary),2) as avg_salary
       FROM EMPLOYEE
       GROUP BY Dno
       HAVING AVG(Salary) > 32000;
```

Output :

Dno	avg_salary
1	55000.00
5	33250.00

Example: Find the maximum of total employee salaries of each department across all departments.

```
Query: SELECT MAX(tot_salary)
       FROM (SELECT Dno, SUM(Salary) FROM EMPLOYEE GROUP BY Dno)
       AS dept_total(Dno, tot_salary);
```

Output:

MAX(tot_salary)
133000.00

WITH clause provides a way of defining a temporary relation whose definition is available only to the query in which the WITH clause occurs.

Example: Find the ssn of the employee who works for the highest number of hours on a particular project. Display the project number and the number of hours the employee has worked as well

Query:
```
WITH max_work(max_hours) AS
    (SELECT MAX(Hours) FROM WORKS_ON)
    SELECT Essn, Pno, Hours
    FROM WORKS_ON, max_work
    WHERE Hours = max_hours;
```

Output:

Essn	Pno	Hours
666884444	3	40.0

Example: Find all departments where the total salary is greater than the average of the total salary at all departments.

Query:
```
WITH dept_total(Dno,tot_salary)
    AS(SELECT Dno, SUM(Salary) FROM EMPLOYEE GROUP BY Dno),
dept_total_avg(avg_tot_salary) AS (SELECT AVG(tot_salary) FROM dept_total)
    SELECT Dname, Dnumber FROM dept_total, dept_total_avg, DEPARTMENT
    WHERE tot_salary > avg_tot_salary AND Dnumber = Dno;
```

Output:

Dname	Dnumber
Research	5

Join Expressions in SQL

Join Operations Overview

Join operations in SQL involve combining two relations to produce another relation as a result. These operations are essentially Cartesian products of tuples from two relations that satisfy certain conditions. The conditions are typically specified through join predicates. Join operations are commonly used as subquery expressions in the `FROM` clause of SQL queries.

There are three main types of joins discussed:

1. **Natural Join**
2. **Inner Join**
3. **Outer Join (including Left, Right, and Full Outer Join)**
4. **Cross Join**

1. Natural Join

- **Definition:** A natural join is an implicit join operation that combines tables based on columns with the same name and data type.
- **Syntax:**

```
SELECT [column_names | *]
FROM table_name1
NATURAL JOIN table_name2;
```

- **Points to Remember:**
  - No need to specify column names for the join.
  - Resultant table always contains unique columns.
  - Possible to perform a natural join on more than two tables.
  - The `ON` clause is not used.

Natural Join Examples

Example 1: List employees along with their department names.

```
SELECT employee_name, employee_salary, Department_name
FROM employee
NATURAL JOIN Department;
```

Example 2: List customers along with their loan amount and address.

```
SELECT CustName, CustID, amount, mobile, address
FROM Borrower
```

```
NATURAL JOIN Loan
NATURAL JOIN cust_info;
```

2. Inner Join

- **Definition:** Inner join combines records from two related tables based on common columns.
- **Syntax:**

```
SELECT table1.column1, table1.column2, table2.column3
FROM table1
INNER JOIN table2 ON table1.columnX = table2.columnX;
```

- **Example: Inner join considering the employee table and department table.**

```
SELECT e.Employee_ID, e.Employee_Name, e.Employee_Salar
y, d.DepartmentID, d.Department_Name
FROM employee e
INNER JOIN department d ON e.Employee_ID = d.Employee_I
D;
```

3. Outer Join

3.1 Left Outer Join

- **Definition:** Left outer join returns all records from the left table and the matched records from the right table. If no match is found in the right table, NULL values are filled in.
- **Syntax:**

```
SELECT table1.column1, table1.column2, table2.column3
FROM table1
```

```
LEFT OUTER JOIN table2 ON table1.columnX = table2.colum
nX;
```

- **Example: Left outer join considering employee table as the left table and department table as the right table.**

```
SELECT e.Employee_ID, e.Employee_Name, e.Employee_Salar
y, d.DepartmentID, d.Department_Name
FROM employee e
LEFT OUTER JOIN department d ON e.Employee_ID = d.Emplo
yee_ID;
```

3.2 Right Outer Join

- **Definition:** Right outer join is similar to left outer join, but it returns all records from the right table and the matched records from the left table.
- **Syntax:**

```
SELECT table1.column1, table1.column2, table2.column3
FROM table1
RIGHT OUTER JOIN table2 ON table1.columnX = table2.colu
mnX;
```

- **Example: Right outer join considering employee table as the left table and department table as the right table.**

```
SELECT e.Employee_ID, e.Employee_Name, e.Employee_Salar
y, d.DepartmentID, d.Department_Name
FROM employee e
RIGHT OUTER JOIN department d ON e.Employee_ID = d.Empl
oyee_ID;
```

3.3 Full Outer Join

- **Definition:** Full outer join returns all records when there is a match in either the left or the right table. If there's no match, NULL values are filled in.
- **Example: Full outer join considering employee table as the left table and department table as the right table.**

```
SELECT e.Employee_ID, e.Employee_Name, e.Employee_Salar
y, d.DepartmentID, d.Department_Name
FROM department d
LEFT OUTER JOIN employee e ON e.Employee_ID = d.Employe
e_ID
UNION
SELECT e.Employee_ID, e.Employee_Name, e.Employee_Salar
y, d.DepartmentID, d.Department_Name
FROM department d
RIGHT OUTER JOIN employee e ON e.Employee_ID = d.Employ
ee_ID;
```

Roles

- Permissions Aggregation: Roles allow you to aggregate a collection of permissions or
  privileges into a single entity. For example, you can create a role called "Manager" and
  assign it permissions to read, write, and delete data in specific database tables.
- Simplify User Management: Roles simplify user management, especially in environments with many users and complex permission structures. Instead of granting
  and revoking permissions for each user individually, you can manage permissions at
  the role level.

- Granting and Revoking Roles: You can grant a role to a user using SQL commands
  like GRANT. Conversely, you can revoke a role using the REVOKE command. When a
  user is granted a role, they inherit the permissions associated with that role

Here are some key points about roles in SQL:

- Hierarchical Roles: In some database management systems, roles can be organized
  hierarchically, where a role can contain other roles. This hierarchical structure can make
  it easier to manage complex permission schemes.
- Dynamic and Static Roles: Some database systems support both dynamic and static
  roles. Dynamic roles are temporary and are assigned when a user logs in or during a
  session. Static roles are assigned explicitly and persist until they are revoked. R
- Role-Based Access Control (RBAC): Roles are often used in the implementation of
  Role-Based Access Control (RBAC) systems, which are designed to provide finegrained control over who can access specific database objects and perform actions on
  them

Role Creation and Assignment

Roles in SQL can be created using the `CREATE ROLE` statement. Once created, privileges can be granted to roles similar to how they are granted to individual users.

Example: Creating a role and granting SELECT privilege:

```
-- Creating a role
CREATE ROLE Assistant_employee;
```

```
-- Granting SELECT privilege to the role
GRANT SELECT ON courses TO Assistant_employee;
```

In this example, a role named `Assistant_employee` is created, and the `SELECT` privilege on the `courses` table is granted to this role.

Role Assignment to Users

Roles can be granted to users, allowing users to inherit the privileges associated with the roles. This simplifies the process of managing permissions for multiple users.

Example: Granting a role to a user:

```
-- Creating a role
CREATE ROLE MD;

-- Granting the role to a user
GRANT MD TO user_name;
```

In this example, a role named `MD` is created, and the role is granted to a specific user (`user_name`). The user now inherits the privileges associated with the `MD` role.

Revoking Privileges from Roles

Privileges granted to roles can be revoked if needed. The `REVOKE` statement is used for this purpose.

Example: Revoking privileges from a role:

```
-- Revoking privileges from a role
REVOKE INSERT, UPDATE, DELETE ON database_name FROM role_nam
e;
```

In this example, the `INSERT`, `UPDATE`, and `DELETE` privileges are revoked from a role (`role_name`) on a specific database (`database_name`).

Removing Roles

Roles can be removed from the system using the `DROP ROLE` statement.

Example: Removing roles:

```
-- Dropping a single role
DROP ROLE role_name;

-- Dropping multiple roles
DROP ROLE role1, role2;
```

In these examples, the `DROP ROLE` statement is used to remove roles from the system. The first example removes a single role (`role_name`), while the second example removes multiple roles (`role1` and `role2`).

Checking Grants

To check the grants for users and roles, the `SHOW GRANTS` statement can be used.

Example: Checking grants for a user:

```
-- Showing grants for a user
SHOW GRANTS FOR root@localhost;

-- Showing grants for a user using a role
SHOW GRANTS FOR root@localhost USING Assistant_employee;
```

In these examples, the `SHOW GRANTS` statement is used to display the grants for a specific user (`root@localhost`). The second example shows grants for a user using a specific role (`Assistant_employee`).

Roles play a crucial role in simplifying the management of privileges and access control in a database system. They provide a way to organize and grant permissions in a more structured and scalable manner.

Functions

Syntax:

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

Example:

```
CREATE FUNCTION Dept_size(deptno INT)
        RETURNS VARCHAR(7)
        BEGIN
                DECLARE No_of_emps INT;
                SELECT COUNT(*) INTO No_of_emps FROM EMPLOYEE WHERE Dno = deptno;
                IF No_of_emps > 3 THEN    RETURN 'HUGE';
                ELSEIF No_of_emps > 2 THEN    RETURN 'LARGE';
                ELSEIF No_of_emps > 1 THEN    RETURN 'MEDIUM';
                ELSE    RETURN 'SMALL';
                END IF;
END
```

Query: select Dname,Dnumber,Dept_size(Dnumber)from department;

Output:

Dname	Dnumber	Dept_size(Dnumber)
Administration	4	HUGE
Headquarters	1	SMALL
Research	5	HUGE

Example: Write a function to find the age of employes

```
CREATE FUNCTION no_of_years(date1 date)
RETURNS INTEGER deterministic
        BEGIN
                DECLARE date2 DATE;
                Select current_date()into date2;
                RETURN year(date2)-year(date1);
        END
```

Query: select Fname,no_of_years(Bdate) from employee;

Output:

Fname	no_of_years(Bdate)
John	58
Franklin	68
Joyce	51
Richard	61
Ramesh	61
James	86
Jennifer	82
Ahmed	54
Alicia	55

Procedure

A procedure (often called a stored procedure) is a collection of pre-compiled SQL statements stored inside the database.

• It is a subroutine or a subprogram in the regular computing language.

• A procedure always contains a name, parameter lists, and SQL statements. We can invoke the
procedures by using triggers, other procedures and applications such as Java , Python , PHP , etc.

```
DELIMITER &&
CREATE PROCEDURE procedure_name [[IN | OUT | INOUT] parameter_name datatype
        [, parameter        datatype]) ]
        BEGIN
                Declaration_section
                Executable_section
        END &&
DELIMITER ;
```

Stored procedures are useful in the following circumstances:

• If a database program is needed by several applications, it can be stored at the server
and invoked by any of the application programs.

•

This reduces duplication of effort and improves software modularity.

• Executing a program at the server can reduce data transfer and communication cost
between the client and server in certain situations.

• These procedures can enhance the modeling power provided by views by allowing
more complex types of derived data to be made available to the database users via the
stored procedures.

Modes of Procedure Parameters:

• IN: It is the default mode. **It takes a parameter as input, such as an attribute**. When we define it, the calling program has to pass an argument to the stored procedure. This parameter's value is always protected.

• OUT: It is used to **pass a parameter as output**. Its value can be changed inside the stored procedure, and the changed (new) value is passed back to the calling program. It is noted that a procedure cannot access the OUT parameter's initial value when it starts.

• INOUT: It is a **combination of IN and OUT parameters.** It means the calling program can pass the argument, and the procedure can modify the INOUT parameter, and then passes the new value back to the calling program.

student_id	student_code	student_name	subject	marks	phone
1	101	Mark	English	68	3454569353
2	102	Joseph	Physics	70	9876543565
3	103	John	maths	70	9765326975
4	104	Barack	maths	90	8769875325
5	105	Rinky	maths	85	6753157975
6	106	Adam	Science	92	7964225686
7	107	Andrew	Science	83	5674243757
8	108	Brayan	Science	85	7523416567
10	110	Alexandar	Biology	67	2347346438

Procedure without ParameterSuppose we want to display all records of this table whose marks are greater than 70 and count all the table rows.

The following code creates a procedure named get_merit_students:

```
DELIMITER $$
CREATE DEFINER=`root`@`localhost` PROCEDURE`get_merit_student`()
BEGIN
        SELECT * FROM student WHERE marks > 70;
        SELECT COUNT(student_code) AS Total_Student FROM student;
END$$
DELIMITER ;
```

Output:

Total_Student
9

Example: Suppose we want to display all records of this table whose marks are greater than 70 and count all the table rows.

The following code creates a procedure named get_merit_students:

```
DELIMITER $$
CREATE PROCEDURE`get_merit_student`()
BEGIN
        SELECT * FROM student WHERE marks > 70;
        SELECT COUNT(student_code) AS Total_Student FROM student;
END$$
DELIMITER ;
```

Output:

Total_Student
9

Query: CALL get_merit_student();

Procedures with IN Parameter

In this procedure, we have used the IN parameter as **'var1'** of integer type to accept a number fro users. Its body part fetches the records from the table using a **SELECT** statement and returns on those rows that will be supplied by the user. It also returns the total number of rows of the specifie table. See the procedure code:

Output:

student_id	student_code	student_name	subject	marks	phone
1	101	Mark	English	68	3454569353
2	102	Joseph	Physics	70	9876543565
3	103	John	maths	70	9765326975

```
DELIMITER &&
CREATE PROCEDURE get_student (IN var1 INT)
BEGIN
   SELECT * FROM student_info LIMIT var1;
   SELECT COUNT(stud_code) AS Total_Student FROM student_info;
END &&
DELIMITER ;
```

Procedures with OUT Parameter

In this procedure, we have used the OUT parameter as the **'highestmark'** of integer type. Its body part fetches the maximum marks from the table using a **MAX() function**. See the procedure code:

```
DELIMITER &&
CREATE PROCEDURE `display_max_mark` (OUT highestmark INTEGER)
BEGIN
   SELECT max(marks) INTO highestmark FROM student;
END&&
DELIMITER ;
```

Output:

@output
92

Query:

```
call display_max_mark(@output);
select @output;
```

Procedures with INOUT Parameter:

In this procedure, we have used the INOUT parameter as **'var1'** of integer type. Its body part first fetches the marks from the table with the specified **id** and then stores it into the same variable var1. The var1 first acts as the IN parameter and then OUT parameter. Therefore, we can call it the INOUT parameter mode. See the procedure code:

```
DELIMITER &&
CREATE PROCEDURE `display_marks` (INOUT var1 INTEGER)
BEGIN
   SELECT marks INTO var1 FROM student WHERE stud_id = var1;
END&&
DELIMITER ;
```

Stored Procedure	Function
Supports in, out and in-out parameters,i.e., input and output parameters	Supports only input parameters, no output parameters.
Stored procedures can call functions as needed	The function cannot call a stored procedure
There is no provision to call procedures from select/having and where statements	You can call functions from a select statement
Transactions can be used in stored procedures	No transactions are allowed
Can do exception handling by inserting try/catch blocks	No provision for explicit exception handling
Need not return any value	Must return a result or value to the caller
All the database operations like insert, update, delete can be performed	Only select is allowed

Cursor

- A database cursor is a mechanism that enables traversal over the records in a database.

- Cursors facilitate subsequent processing in conjunction with the traversal, such as retrieval, addition and removal of database records.
- A cursor in SQL is a temporary work area created in system memory when a SQL
  statement is executed.

Attribute	Description
%FOUND	It will return TRUE in case an INSERT, UPDATE, or DELETE statement affects one or more rows or a SELECT INTO statement returns one or more rows. In other cases, it will return FALSE.
%NOTFOUND	It is technically the opposite of %FOUND attribute. It returns TRUE in case an INSERT, UPDATE, or DELETE statement doesn't affect any rows or a SELECT INTO statement returns no rows. Else it returns just FALSE.
%ISOPEN	This attribute will always return FALSE for implicit cursors as the SQL cursor is automatically closed immediately after the associated SQL statement is executed.
%ROWCOUNT	It returns the total number of affected rows by an INSERT, UPDATE, or DELETE statement, or the rows returned by a SELECT INTO statement.

Implicit Cursors

- Auto-created by Oracle when SQL is executed if there is no explicit cursor used.
- Users or programmers cannot control the information or programs in it
- Associated with INSERT, UPDATE and DELETE types of DML operation statements
- Attributes: SQL%FOUND, SQL%NOTFOUND, %ISOPEN, %ROWCOUNT

Explicit Cursors

- User-defined cursors which help to gain more control over the context part.
- It is defined in the declaration area of the SQL block.
- Created on SELECT statements that returns multiple records.
- Attributes: SQL%FOUND, SQL%NOTFOUND, %ISOPEN, %ROWCOUNT.

Properties:

• READ ONLY :Using these cursors you cannot update any table.
• Non-Scrollable: Using these cursors you can retrieve records from a table in one direction
i.e., from top to bottom.
• Asensitive : These cursors are insensitive to the changes that are made in the table i.e. the
modifications done in the table are not reflected in the cursor,

UNIVE

Syntax:

```
DECLARE cursor_name CURSOR FOR select_statement
OPEN cursor_name
FETCH [[NEXT] FROM] cursor_name INTO var_name [, var_name] ...
CLOSE cursor_name
```

When the FETCH statement is called, the next row is read in the result set each time.
But a time comes when it reaches to the end of the set and no data is found there, so to handle this condition with MYSQL cursor we need to use a NOT FOUND handler.

Syntax:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET variable_name = 1;
```

Trigger

- A trigger is a statement that the system executes automatically as a side effect of a
  modification to the database.

Defining a trigger involves:

- Specifying when a trigger needs to be executed. This consists of two components:

Event – which causes the trigger to be checked
Condition – which must be satisfied for trigger execution to proceed

- Specifying the action to be taken when the trigger executes
- Once we enter a trigger into the database, the system executes it whenever the specified
  event occurs and the corresponding condition is satisfied

Usecases of triggers

- Triggers can be used to implement certain integrity constraints that cannot be specified using
  the constraint mechanism of SQL.
- Triggers are also useful mechanisms for alerting humans or for starting certain tasks
  automatically when certain conditions are met.

Syntax:

```
CREATE TRIGGER trigger_name
(AFTER | BEFORE) (INSERT | UPDATE | DELETE) ON table_name
FOR EACH ROW
BEGIN
  --variable declarations
  --trigger code
END;
```

Let's try to understand the syntax of the CREATE TRIGGER statement:

CREATE TRIGGER trigger_name

- The first line of the syntax (shown above) is used to **specify the name of the trigger.**

(AFTER | BEFORE) (INSERT | UPDATE | DELETE) ON table_name

- The second line of the syntax is used to specify the **event which causes the trigger to be checked**.

➢ table_name - specifies the **relation** which the trigger must check for modification

➢ INSERT | UPDATE | DELETE - specifies the **type of modification to be checked** i.e. whether the trigger must be checked on insertion, updation or deletion of tuples.

➢ AFTER | BEFORE - specifies **when the trigger must be checked** i.e. before or after the specified relation undergoes modification.

The entries in the tables are shown below:

Student_sample:

SRN	Name
1	Harry
2	Ron
3	Hermione
4	Draco

Marks_sample:

SRN	Course	Marks
1	C01	98
1	C02	89
2	C01	100
2	C02	98
2	C03	80
3	C01	95
3	C02	98

Note that the Marks_history table is initially empty.

Example: Create a trigger that checks if the entered marks are valid or not whenever an entry is made in the Marks_sample table.

Code:

```
DELIMITER //
CREATE TRIGGER CheckMarks
BEFORE INSERT ON Marks_sample
FOR EACH ROW
BEGIN
        IF NEW.marks < 0 OR NEW.marks > 100 THEN
                SIGNAL SQLSTATE '45000'
                SET MESSAGE_TEXT = 'Invalid marks: Marks must be between 0 and 100';
        END IF;
END;//
DELIMITER ;
```

Example: Implement a trigger that stores all tuples deleted from the *Marks_sample* table in the *Marks*_history table

Code:

```
DELIMITER //
CREATE TRIGGER backup_marks_info
BEFORE DELETE
ON Marks_sample
FOR EACH ROW
BEGIN
INSERT INTO Marks_history SELECT * FROM Marks_sample WHERE SRN = OLD.SRN AND COURSE =OLD.COURSE;
END;//
DELIMITER ;
```

Recursive Query

A recursive query, often used in the context of relational databases and SQL, involves a query that references itself in order to retrieve hierarchical or recursive information. This type of query is useful when dealing with data that has a

hierarchical structure, such as organizational charts, file systems, or hierarchical categorizations.

In SQL, a common way to implement recursive queries is through the use of the `WITH RECURSIVE` clause. Here's a simple example to illustrate the concept. Let's consider a table representing an organizational hierarchy:

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(100),
    manager_id INT
);

INSERT INTO employees VALUES (1, 'John', NULL);
INSERT INTO employees VALUES (2, 'Alice', 1);
INSERT INTO employees VALUES (3, 'Bob', 1);
INSERT INTO employees VALUES (4, 'Charlie', 2);
INSERT INTO employees VALUES (5, 'David', 2);
```

Now, let's use a recursive query to retrieve the hierarchical structure:

```
WITH RECURSIVE EmployeeHierarchy AS (
    SELECT employee_id, employee_name, manager_id
    FROM employees
    WHERE manager_id IS NULL

    UNION ALL

    SELECT e.employee_id, e.employee_name, e.manager_id
    FROM employees e
    INNER JOIN EmployeeHierarchy eh ON e.manager_id = eh.empl
oyee_id
)

SELECT * FROM EmployeeHierarchy;
```

In this example, the recursive query starts with selecting employees who don't have managers (i.e., those with `manager_id` set to `NULL`). The `UNION ALL` operator is then used to combine the base case with the recursive case, which involves joining the employees table with the already selected rows in the `EmployeeHierarchy` common table expression (CTE).

As a result, the query returns the entire organizational hierarchy, showing the relationships between employees and their managers.

The syntax for recursive queries can vary slightly depending on the database management system (DBMS) you are using. However, I'll provide a general overview using the common syntax found in databases that support recursive queries, such as PostgreSQL and MySQL.

The `WITH RECURSIVE` clause is commonly used to define a recursive common table expression (CTE). Here's the basic syntax:

```
WITH RECURSIVE cte_name (column1, column2, ...) AS (
    -- Anchor member (non-recursive term)
    SELECT column1, column2, ...
    FROM some_table
    WHERE condition

    UNION [ALL]

    -- Recursive member
    SELECT column1, column2, ...
    FROM some_table
    JOIN cte_name ON join_condition
)
SELECT * FROM cte_name;
```

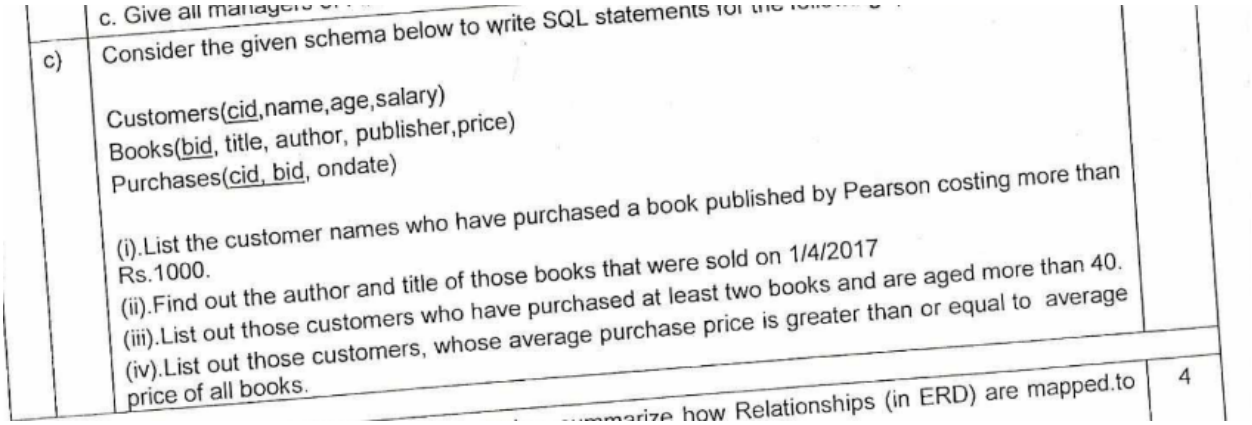
Let's break down the syntax components:

- `WITH RECURSIVE`: Indicates the beginning of a recursive CTE.
- `cte_name`: The name of the recursive CTE.

- `(column1, column2, ...)`: Optional column list specifying the columns in the CTE.
- `SELECT column1, column2, ... FROM some_table WHERE condition`: The anchor member (non-recursive term) that serves as the starting point for the recursion.
- `UNION [ALL]`: Combines the anchor member with the recursive member. `ALL` includes duplicate rows; omitting it removes duplicates.
- `SELECT column1, column2, ... FROM some_table JOIN cte_name ON join_condition`: The recursive member that references the CTE itself in the `JOIN` clause.
- `SELECT * FROM cte_name`: The final `SELECT` statement retrieves the results of the recursive CTE.

It's essential to have a proper termination condition in the recursive member to prevent an infinite loop. This often involves referencing the CTE columns in the `WHERE` clause to break the recursion.

Examples



c. Give all managers

c) Consider the given schema below to write SQL statements for the

Customers(cid,name,age,salary)
Books(bid, title, author, publisher,price)
Purchases(cid, bid, ondate)

(i).List the customer names who have purchased a book published by Pearson costing more than Rs.1000.
(ii).Find out the author and title of those books that were sold on 1/4/2017
(iii).List out those customers who have purchased at least two books and are aged more than 40.
(iv).List out those customers, whose average purchase price is greater than or equal to average price of all books.

4

summarize how Relationships (in ERD) are mapped.to

Certainly! Based on the given schema, here are the SQL statements for the specified queries:

(i) List the customer names who have purchased a book published by Pearson costing more than $50:

```
SELECT DISTINCT C.name
FROM Customers C
JOIN Purchases P ON C.cid = P.cid
JOIN Books B ON P.bid = B.bid
WHERE B.publisher = 'Pearson' AND B.price > 50;
```

(ii) Find out the author and title of those books that were sold on 1/4/2017:

```
SELECT DISTINCT B.author, B.title
FROM Books B
JOIN Purchases P ON B.bid = P.bid
WHERE P.ondate = '2017-04-01';
```

(iii) List out those customers who have purchased at least two books and are aged more than 40:

```
SELECT C.name
FROM Customers C
JOIN Purchases P ON C.cid = P.cid
GROUP BY C.pItname, C.age
HAVING COUNT(P.bid) >= 2 AND C.age > 40;
```

(iv) List out those customers whose average purchase price is greater than or equal to average salary:

```
SELECT C.name
FROM Customers C
JOIN Purchases P ON C.cid = P.cid
GROUP BY C.name, C.salary
HAVING AVG(P.price) >= C.salary;
```

Note: The exact syntax may vary slightly depending on the SQL database system you are using. The above examples assume a standard SQL syntax.

:

3.		For the schema, employee(emp_name, city) company(comp_name, city) works(emp_name, comp_name) manages(emp_name, manager_name) both emp_name and manager_name references emp_name	
	a	Write the SQL queries for the following requirements i. List the employees who live in the same city as the company they work for. Display employee name, company name and the city. ii. List the colleagues of 'Durga'. (Colleagues are people who work in the same company) iii. List the employee names and the city for employees who do not have a manager. **Use Correlated Nested Query.**	6
	b	Write the queries in SQL for: i. List all the employees (Display employee name, city and manager name). If the employee has a manager, display the manager name otherwise just display the employee name and city. ii. List the managers and the number (count) of employees under them. Ignore managers managing only one person. Order the output such that the manager with most people under him appears in the first row.	4

		For the schema, book(book_id, title, publisher_name, no_of_copies) book_author(book_id, author_name) book_borrowals(book_id, card_id, borrowal_date, due_date, return_date) borrower(card_id, name, address, phone, email)	
	c	Give the relational algebra queries for: i. List all the books (book_id) borrowed by the member with name 'Dhatri' ii. List the borrowers (name) who have borrowed books and not returned them. (borrowals have a NULL value in return_date) iii. List the borrowers (name) who have not borrowed any books ever.	6
	d	Write the queries in relational algebra for: i. For each book, display the total number of times it has been borrowed. ii. List the borrowers (name) who have read all the books by 'R.K Narayan.'	4

SQL Queries for Employee Schema

Part a:

1. List employees living in the same city as their company:

```
SELECT e.emp_name
FROM employee e, company c
WHERE e.city = c.comp_city AND e.emp_name = c.comp_name;
```

1. List colleagues of 'Durga':

```
SELECT e1.emp_name
FROM works w1, works w2, employee e1, employee e2
WHERE w1.comp_name = w2.comp_name AND e1.emp_name = w1.emp_na
me AND e2.emp_name = w2.emp_name AND e2.emp_name = 'Durga' AN
D e1.emp_name != 'Durga';
```

1. List the employee names and the city for employees who do not have a manager:

```
SELECT e.emp_name, e.city
FROM employee e
WHERE NOT EXISTS (
    SELECT *
    FROM manages m
    WHERE m.emp_name = e.emp_name
);
```

Part b:

1. List all the employees with optional manager name:

```
SELECT e.emp_name, e.city, m.manager_name
FROM employee e
LEFT JOIN manages m ON e.emp_name = m.emp_name;
```

1. List the managers and the number (count) of employees under them, ignoring managers managing only one person:

```
SELECT m.manager_name, COUNT(*) as num_employees
FROM manages m
GROUP BY m.manager_name
HAVING COUNT(*) > 1
ORDER BY num_employees DESC;
```

Relational Algebra Queries for Book Schema

Part c:

1. List all the books (\(book_id \)) borrowed by the member with name 'Dhatri':

   $\pi_{bookid}(\sigma name =' Dhatri'(borrower) \bowtie book\_borrowals)$

2. List the borrowers (\(name \)) who have borrowed books and not returned them:
3. $\pi_{name}(\sigma_{return\_date=NULL}(book\_borrowals) \bowtie borrower)$
4. List the borrowers (\(name \)) who have not borrowed any books ever:
5. $\pi_{name}(borrower) - \pi_{name}(borrower \bowtie book\_borrowals)$

Part d:

1. For each book, display the total number of times it has been borrowed:
2. $\gamma_{book\_id;COUNT(book\_id)\rightarrow total\_borrows}(book\_borrowals)$
3. List the borrowers (\(name \)) who have read all the books by 'K.R. Narayan':
4. $\pi_{name}(\sigma_{author\_name='K.R.Narayan'}(book\_author) \bowtie book\_borrowals \bowtie borrower)$