Semi Join and Semi-difference (IN and NOT IN of SQL)

Because of union compatibility requirement, direct use of SET operations in SQL is limited.

- INTERSECT is accomplished by NATURAL JOIN or SEMI JOIN/SEMI INTERSECT (IN in SQL)
- EXCEPT typically is accomplished by SEMI Difference (NOT IN of SQL)
- In some cases UNION can be performed by having OR in tuple SELECTION criteria (in WHERE Clause of SQL). For example
  - Students either study in BCS or BIT

```
select * from student
    where progid='BCS' or progid='BIT'
```

    OR

```
select * from student where progid='BCS'
UNION
select * from student where progid='BIT'
```

  - However, this is typically possible only when we have union based on different options on same relation or so. For example, following UNION may not be possible through OR-ing strategy-

```
select superssn from employee
UNION
select mgrssn from department
```

Note: DISTINCT is implicit in SET operations in SQL

JOIN is basically INTERSECTION problem

JOIN can be understood as: join produces "joined" tuples of "common tuples" in operand relations; and "commonness" is checked based on "join condition".

Semi JOIN or Semi Intersection [SQL IN]

Semi Intersection

Let Semi-Intersection be expressed as following

$$r1 \frac{SEMI\_INTERSECTION}{r1.A = r2.B} r2$$

where A and B are same size attribute sets.

In plain words, it can be stated as following

$$r1 \frac{MATCHING}{mactching\ cond\ is\ (r1.A = r2.B)} r2$$

It is algebraically equivalent to following

$$r1 * \left(\pi_A(r1) \cap \pi_B(r2)\right)$$

where * stands for Natural Join.

Above SEMI INTERSECTION is also algebraically equivalent to following

$$\pi_{r1.*}(r1 \underset{r1.A=r2.B}{\overset{\bowtie}{\text{———}}} r2)$$

This means we are performing join but are interested in values of left relation r1 only.

Therefore SEMI INTERSECTION is, more popularly called as "SEMI JOIN"

Semi Intersection/Join in SQL

SQL provides IN keyword for performing semi intersection. Above algebraic SEMI INTERSECTION or SEMI JOIN can be expressed as

```
SELECT * FROM
WHERE A1, A2, … Am IN (SELECT B1, B2, .. Bm FROM r2);
```

Exercise ##:

Compute employees (i.e., employee.*) that are both (manager of some department and supervisor of some employee)

#using SET INTERSECTION (Algebra)

$$r1 \leftarrow \pi_{superssn}(EMPLOYEE) \cap \pi_{mgrssn}(DEPARTMENT)$$

$$result \leftarrow \pi_{e.*}\left(r1 \overset{\bowtie}{\boxed{e.ssn = r1.superssn}} EMPLOYEE_e\right)$$

OR

$$r1(ssn) \leftarrow \pi_{superssn}(EMPLOYEE) \cap \pi_{mgrssn}(DEPARTMENT)$$

$$result \leftarrow r1 * EMPLOYEE$$

#using SET INTERSECTION (SQL)

```
SELECT e.* from employee as e join
    (select superssn from employee
    intersect select mgrssn from department) as r1
    on r1.superssn = e.ssn;
```

#using SEMI JOIN/INTERSECTION (Algebra)

$$r1 \leftarrow EMPLOYEE_e \frac{SEMI\_INTERSECTION}{e.superssn = d.mgrssn} DEPARTMENT_d$$

$$result \leftarrow EMPLOYEE_e \frac{SEMI\_INTERSECTION}{e.ssn = r1.superssn} r1$$

OR

$$r1 \leftarrow \pi_{superssn}(EMPLOYEE) \cap \pi_{mgrssn}(DEPARTMENT)$$

$$result \leftarrow EMPLOYEE_e \frac{SEMI\_INTERSECTION}{e.ssn = r1.superssn} r1$$

#using SEMI JOIN/INTERSECTION (SQL) using IN

```
SELECT * from employee where ssn IN (select superssn from employee
     intersect select mgrssn from department);
```

OR using nested semi intersection using IN in SQL-

```
SELECT * from employee
    where ssn IN (select superssn from employee
                    where superssn IN (select mgrssn from department));
```

Exercise ##:

Compute “employees (i.e., employee.*) that work on some project”

$$result \leftarrow EMPLOYEE_e \frac{SEMI\_JOIN}{e.ssn = w.essn} WORKS\_ON_w$$

Semi DIFERENCE [SQL NOT IN]

Let Semi-Difference be expressed as following

$$r1 \frac{SEMI\_DIFFERENCE}{r1.A = r2.B} r2$$

where A and B are same size attribute sets.

In plain words, it can be stated as following

$$r1 \frac{NOT\ MATCHING}{mactching\ cond\ is\ (r1.A = r2.B)} r2$$

It is algebraically equivalent to following

$$r1 * (\pi_A(r1) - \pi_B(r2))$$

where * stands for Natural Join.

Example

Compute employees (i.e., employee.*) that are not managers

#using SET DIFFERENCE (Algebra)

$$r1 \leftarrow \pi_{ssn}(EMPLOYEE) - \pi_{mgrssn}DEPARTMENT$$

$$result \leftarrow EMPLOYEE * r1$$

#using SET DIFFERENCE (SQL)

```
SELECT e.* from employee as e natural join
    (select ssn from employee
    EXCEPT select mgrssn from department) as r1;
```

#using SEMI DIFFERENCE (Algebra)

$$result \leftarrow EMPLOYEE_e \frac{SEMI\_DIFFERENCE}{e.ssn = d.mgrssn} DEPARTMENT_d$$

#using SEMI DIFFERENCE (SQL) - using NOT IN

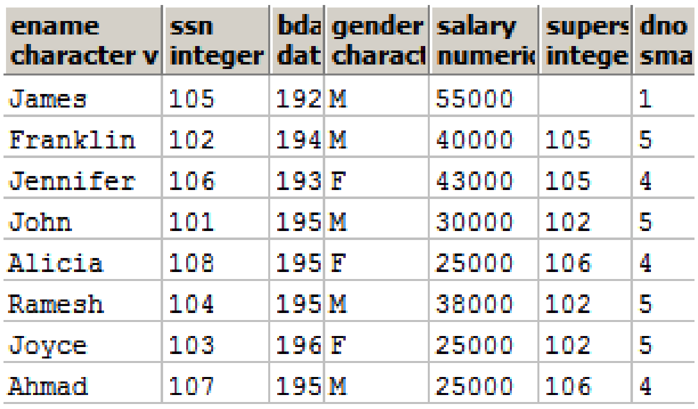
```
SELECT * from employee WHERE ssn NOT IN
    (select mgrssn from department);
```

Exercise ##:

Compute employees (i.e., employee.*) that are supervisors (supervises some employees) but not managers (that are manager of any department)

Aggregate operations on Relations

Given a relation, we perform some operations over all tuples or grouped tuples



ename character v	ssn integer	bda dat	gender charact	salary numeric	supers intege	dno sma
James	105	192	M	55000		1
Franklin	102	194	M	40000	105	5
Jennifer	106	193	F	43000	105	4
John	101	195	M	30000	102	5
Alicia	108	195	F	25000	106	4
Ramesh	104	195	M	38000	102	5
Joyce	103	196	F	25000	102	5
Ahmad	107	195	M	25000	106	4

Aggregation operations are basically counting or summing of an attribute-values. Following are common aggregation operations – COUNT, SUM, AVG, MAX, MIN for columns of a table.

Aggregation can be computed over all tuples or on grouped tuples of relations.

Following are examples of queries that require performing aggregate operations, and thus are called aggregate queries -

- Find out total salary we pay to all employees.
- Find out total number of employees, total number of supervisors, and so
- Find out employee who are drawing maximum salary
- Find out employee who are drawing less than average salary,
- Find out average salary paid to managers,
- And so forth

Aggregation operations are expressed using script F (F) operator

$\mathcal{F}_{<funtion\text{-}list>}(r)$

Examples

$\mathcal{F}_{COUNT(SSN),\ AVG(SALARY)}$ (employee)

Written in SQL as

SELECT count(ssn), avg(salary) FROM employee;

The result of this operation will be a single tuple having two columns?

Another example

$\mathcal{F}_{\text{COUNT(SSN), MAX(SALARY), MIN(SALARY), AVG(SALARY)}}$ (employee)

SELECT count(ssn), max(salary), min(salary), avg(salary) FROM employee;

Aggregation over grouped tuples

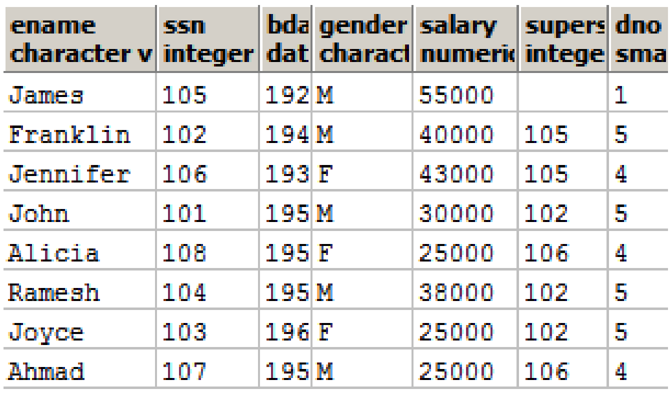
In this case tuples of operand relation are grouped based on some attributes(s) value(s). We call these attributes as grouping attributes.

For every distinct value of grouping attribute(s), there will be a group, and then aggregation operation is computed for each group.

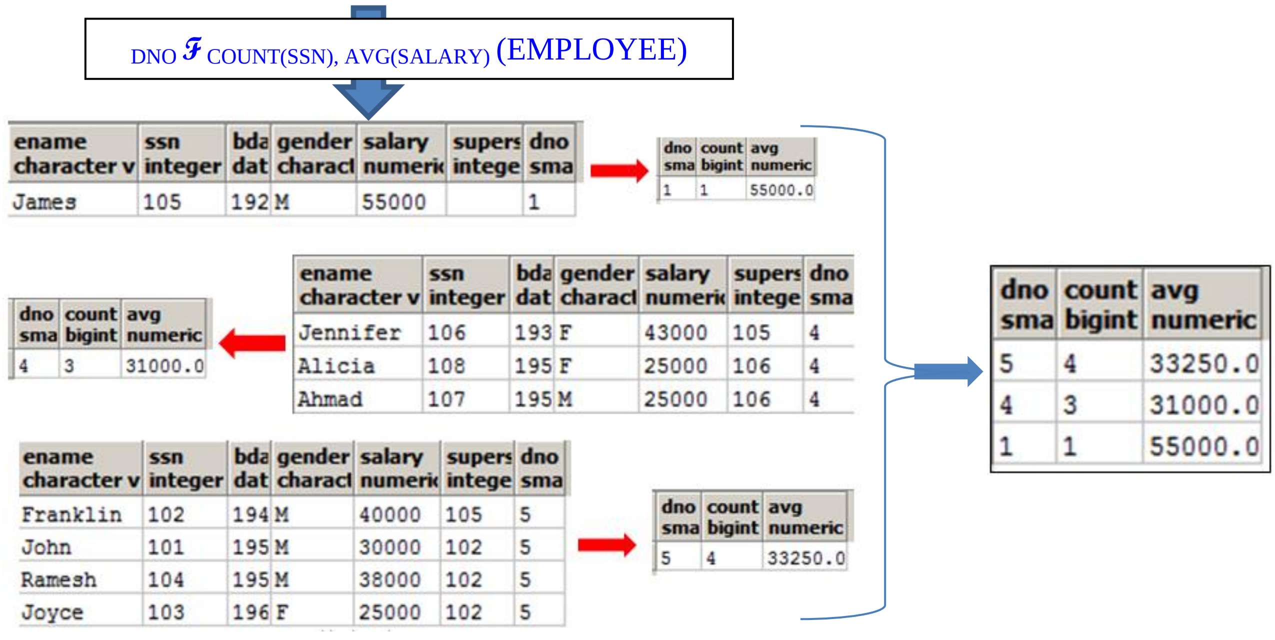
${}_{\text{<grouping-attributes>}}\mathcal{F}_{\text{<funtion-list>}}$(r)

Example:

${}_{\text{DNO}}\mathcal{F}_{\text{COUNT(SSN), AVG(SALARY)}}$ (EMPLOYEE)



ename	ssn	bda	gender	salary	supers	dno
character v	integer	dat	charact	numeric	intege	sma
James	105	192	M	55000		1
Franklin	102	194	M	40000	105	5
Jennifer	106	193	F	43000	105	4
John	101	195	M	30000	102	5
Alicia	108	195	F	25000	106	4
Ramesh	104	195	M	38000	102	5
Joyce	103	196	F	25000	102	5
Ahmad	107	195	M	25000	106	4



ename	ssn	bda	gender	salary	supers	dno
character v	integer	dat	charact	numeric	intege	sma
James	105	192	M	55000		1

dno	count	avg
sma	bigint	numeric
1	1	55000.0

ename	ssn	bda	gender	salary	supers	dno
character v	integer	dat	charact	numeric	intege	sma
Jennifer	106	193	F	43000	105	4
Alicia	108	195	F	25000	106	4
Ahmad	107	195	M	25000	106	4

dno	count	avg
sma	bigint	numeric
4	3	31000.0

ename	ssn	bda	gender	salary	supers	dno
character v	integer	dat	charact	numeric	intege	sma
Franklin	102	194	M	40000	105	5
John	101	195	M	30000	102	5
Ramesh	104	195	M	38000	102	5
Joyce	103	196	F	25000	102	5

dno	count	avg
sma	bigint	numeric
5	4	33250.0

dno	count	avg
sma	bigint	numeric
5	4	33250.0
4	3	31000.0
1	1	55000.0

SELECT dno, count(ssn), avg(salary) FROM employee GROUP BY dno;

Another example

$_{\text{DNO, GENDER}}\mathcal{F}_{\text{COUNT(SSN), AVG(SALARY)}}$ (EMPLOYEE)

```
SELECT dno, gender, count(ssn), avg(salary)
        FROM employee GROUP BY dno, gender;
```

Renaming of operations in aggregation

You can have renaming used in aggregate operation,

For example, an aggregate query-

$_{\text{DNO}}\mathcal{F}_{\text{COUNT(SSN)} \rightarrow \text{No\_of\_Emps, AVG(SALARY)} \rightarrow \text{AVG\_SAL}}$ **(EMPLOYEE)**

In SQL, we write as following -

```
SELECT dno, count(ssn) AS No_of_Emps, avg(salary) AS avg_sal
        FROM employee
        GROUP BY dno;
```

NULL's in Aggregation

- NULL never contribute to sum, average, or count, and can never be the minimum or maximum of a column.
- But if there are no non-NULL values in a column, then the result of the aggregation is NULL.

Examples

```
SELECT count(*) FROM employee; -- counts all row, * is used for counting rows!
SELECT count(dno) FROM employee; -- counts occurrence of dno values (excludes NULL)
SELECT count(DISTINCT dno) FROM employee; -- counts of distinct values for dno attribute
SELECT count(superssn) FROM employee;
SELECT count(DISTINCT superssn) FROM employee;
SELECT min(salary), max(salary), avg(salary) FROM employee;
```

Exercise: Can you figure out, why following references (in red) are invalid?

```
SELECT dno, ssn, avg(salary) AS avg_sal FROM employee GROUP BY dno;

SELECT dno, avg(salary) AS avg_sal FROM employee;
```

HAVING clause

HAVING is used to specify restrict over result of aggregation

For example:

```
SELECT dno, avg(salary) AS avg_sal
        FROM employee GROUP BY dno
        HAVING avg(salary) > 50000;
```

Algebraically, equivalent to

$$r1 \leftarrow {}_{DNO}\mathcal{F}_{AVG(SALARY) \rightarrow AVG\_SAL}(EMPLOYEE)$$

$$result \leftarrow \sigma_{avg(sal) > 50000}(r1)$$

Note that you cannot use avg_sal instead of avg(salary) in HAVING clause, because renaming is done at the time of projection.

Semantics of SQL SELECT statement

```
SELECT <attrib and/or function-list> (5)
FROM <relation-expression> (1)
[WHERE <condition>] (2)
[GROUP BY <grouping attribute(s)>] (3)
[HAVING <group-filter-condition>] (4)
[ORDER BY <attrib-list>]; (6)
```

$r1 \leftarrow \text{<relational-expression>}$

$r2 \leftarrow \sigma_{\text{<where-condition>}}(r1)$

$r3 \leftarrow {}_{\text{<group-attributes>}}F_{\text{<aggregate-operation>}}(r2)$

$r4 \leftarrow \sigma_{\text{<group-filter-condition>}}(r3)$

$r5 \leftarrow \pi_{\text{<attrib-list>}}(r4)$

- Result of FROM and WHERE is given to GROUP BY operation
- GROUP BY operation computes *aggregated values* for each group value, and gives you one tuple for each group value.
- **group-filter-conditions** in HAVING are used to apply restriction over result of GROUP BY operation
- Finally project is applied as defined in SELECT clause of SELECT statement.

Exercise: Convert to Relational Algebra

```
SELECT * FROM r1, r2
WHERE ...;
```

r1 and r2 are relational expressions. Therefore writing like below is perfectly fine-

```
SELECT * FROM
(s1 JOIN s2 ON s1.a2 = s2.b1) as r1, r2
WHERE ...;
```

Parenthesis in above expression is optional but makes expression more explicit.
Following is also OK.

```
SELECT * FROM
s1 JOIN s2 ON s1.a2 = s2.b1, r2
WHERE ...;
```