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

Because of type compatibility, use of SET operations in SQL, has limited use in practice

- In many 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 (can solved as UNION or by "OR"
  - Employee that are either manager or supervisor; however cannot be solved by OR

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

- INTERSECT is accomplished by NATURAL JOIN or SEMI JOIN (IN in SQL)
- EXCEPT could be accomplished by SEMI Difference (NOT IN of SQL)

Note: DISTINCT is implied in SET operations in SQL

JOIN can be as INTERSECTION problem

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

Semi JOIN or Semi Intersection [SQL IN]

Semi Intersection

Motivating Example#1

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

Algebra Solution:

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

$$result \leftarrow \pi_{e.*}(EMPLOYEE_{e} * r1)$$

Solution in SQL

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

Compare above SQL solution with following-

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

In algebraic terms, read above SQL expression as intersection but based on specified attributes only.

Say above is intersection of EMPLOYEE and DEPARTMENT on employee's SSN and department's MGRSSN

In algebraic term, we call such an intersection as "SEMI INTERSECTION"; and let us say expressed as

$$result \leftarrow EMPLOYEE_e \xrightarrow[e.ssn\ =\ d.mgrssn]{SEMI\_INTERSECTION} DEPARTMENT_d)$$

Semi intersection is expressed using IN keyword in SQL. It is specified in where clause of left operand relation as following -

```
SELECT * from employee as e
    where e.ssn IN (select d.mgrssn from department as d);
```

General form of SQL IN can be expressed as following

```
SELECT * from r1 where a1,a2,..an IN (select b1, b2, ..bn from r2);
```

Above general SQL expression can be interpreted as following- give tuples from r1 where value-list of <a1,a2,..an> from r1 exists in value list of <a1,a2,..an> from r2.

Exercise:

Express query "employees that work on some project" using INTERSECTION and SEMI-INTERSECTION.

Motivating Example #2

Another example, somewhat complicated; Compute employees (i.e., employee.*) that are both (manager of some department and supervisor of some employee)

Algebra Solution:

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

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

OR

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

$$result \leftarrow \pi_{e.*}(r1 * EMPLOYEE_e)$$

Expressed in SQL

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

Algebra Solution using semi intersection:

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

$$result \leftarrow EMPLOYEE_e \xrightarrow[e.ssn\ =\ r1.superssn]{SEMI\_INTERSECTION} r1$$

Semi solution expressed in SQL using IN

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

One more solution 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));
```

SEMI JOIN

Notice that query "employees (i.e., employee.*) that work on some project" can also be expressed as following-

$$result \leftarrow \pi_{e.*}\left(EMPLOYEE_e \underset{\boxed{e.ssn = w.essn}}{\bowtie} WORKS\_ON_w \right)$$

Can be expressed as

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

It should be easy to understand that SEMI-JOIN and SEMI-INTERSECTION are algebraically same. And in another (English) words both can be understood as "r1 matching with r2 with respect to specified matching condition. For example EMPLOYEE MATCHING WORS_ON with respect to employee's ssn and works_on's essn

Semi DIFERENCE [SQL NOT IN]

Example #1

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

Algebra Solution:

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

$$result \leftarrow \pi_{e.*}(EMPLOYEE_e * r1)$$

Solution in SQL

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

Another solution in SQL (using NOT IN)

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

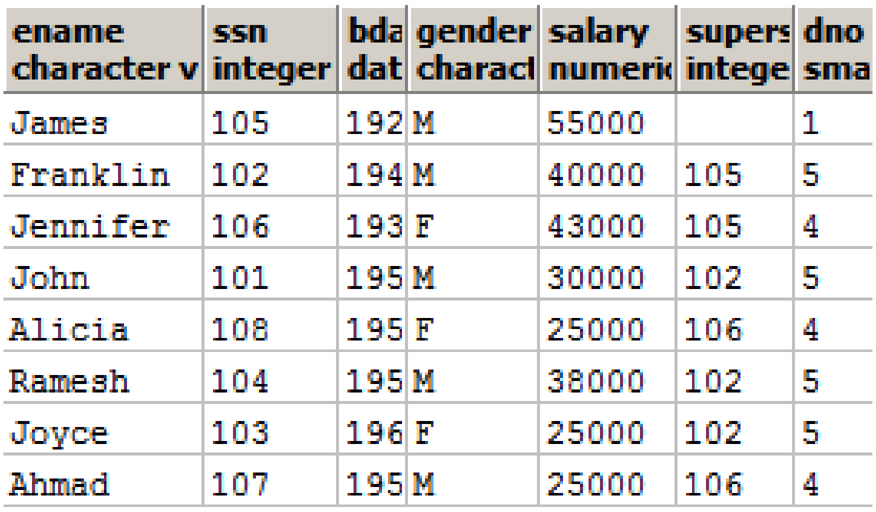
Can be expressed in algebra as SEMI JOIN

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

In another words semi difference can be understood as NOT MATCHING with respect to specified condition. For example, EMPLOYEE NOT MATCHING WORS_ON with respect to employee's ssn and works_on's essn.

Aggregate operations on Relations

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



ename character v	ssn integer	bda dat	gender charact	salary numeri	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

Are basically counting or summing of an attribute-values. Following are common aggregation operations: SUM, AVG, MAX, MIN, COUNT are common aggregate operations over specified column of a table

Aggregation can be done over all tuples or grouped tuples of a relations

Following are examples of aggregate queries-

- Find out total salary we pay
- Find out total number of employees, total number of supervisors
- 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

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

Examples

$\mathcal{F}_{\text{COUNT(SSN), AVG(SALARY)}}$ (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. We call these attributes as grouping attributes.

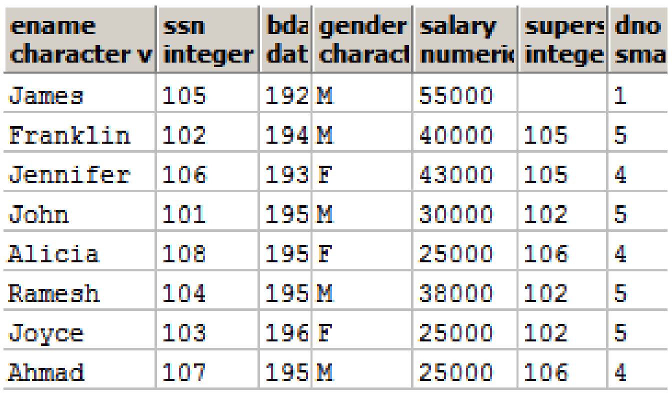
For every distinct value of grouping attribute(s), there will be a group.

Then aggregation operation is applied on each of the group

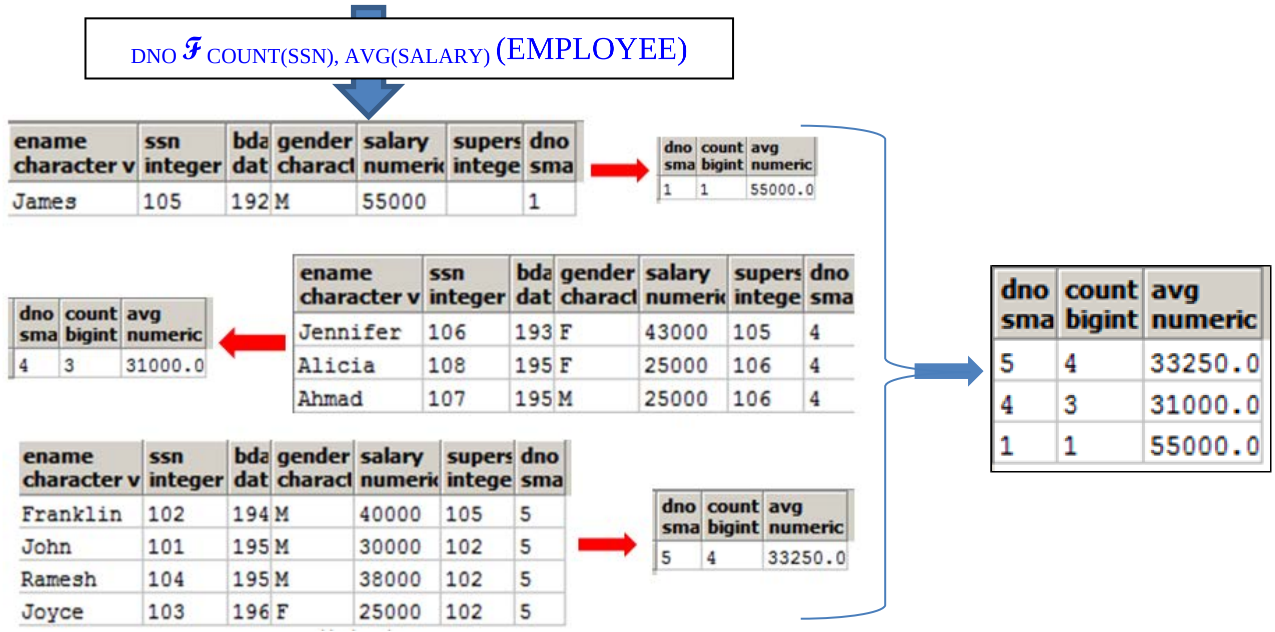
$_{\text{<grouping-attributes>}}\mathcal{F}_{\text{<funtion-list>}}$(r)

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



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

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

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-

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

In SQL, you write, as

```
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;

SELECT count(dno) FROM employee;

SELECT count(DISTINCT dno) FROM employee;

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 ...;
```