

# The Relational Model

Dirk Van Gucht

# The relational model

- A **relational database** is a collection of relations (tables).

Student

Sid	Sname	Major	Byear
s1	John	CS	1990
s2	Ellen	Math	1995
s3	Eric	CS	1990
s4	Ann	Biology	2001

Course

Cno	Cname	Dept
c1	Dbs	CS
c2	Calc1	Math
c3	Calc2	Math
c4	AI	Info

Enroll

Sid	Cno	Grade
s1	c1	B
s1	c2	A
s2	c3	B
s3	c1	A
s3	c2	C

# Terminology: definitions

Attributes	relation column names
Relation schema	consists of a relation name and a set of attributes
Tuples	rows of a relation
Tuple components	a tuple has one component for each attribute of the relation; a tuple component value must be from the domain of the attribute
Relation instance	a finite set of tuples
Relation	a pair of a relation schema and a relation instance

- The schema of a relation is called its **meta-data**.
- The instance of a relation is called its **data**.

# Terminology: example

Student ← relation name

Sid	Sname	Major	Byear	← attributes
s1	John	CS	1990	← tuple
s2	Ellen	Math	1995	← tuple
s3	Eric	CS	1990	← tuple
s4	Ann	Biology	2001	← tuple

- The relation name and header of the table is the **relation schema**.
- The body of the table is the **relation instance**.

# Attribute domains

- Different **types** are possible:
  - **Basic types**: boolean, integer, real, character, text;
  - **Composite types**: monetary, date;
  - **Enumeration types**: list of values, sets, arrays;
  - **Semi-structured types**: XML, JSON;
  - **Arrays**,

and many more!

# Tuples and tuple components

- Consider the `Student(Sid, Sname, Major, Byear)` schema and the **tuple**  $t = (s1, \text{John}, \text{CS}, 1990)$ .
- Then  $t.\text{Sid}$ ,  $t.\text{Sname}$ ,  $t.\text{Major}$ , and  $t.\text{Byear}$  are the **tuple components of t** and their **domain values** are  $s1$ ,  $\text{John}$ ,  $\text{CS}$ , and  $1990$ , respectively.

# Null values

- **NULL** represents a **missing** or an **unknown** value of a tuple (**incomplete information**).
- E.g., **(s5, Marc, NULL, 1995)** indicates that the **Major** component of this tuple is missing or unknown.
- Caveat: multiple occurrences of **NULL** values do not imply that they are the same: they may or may not be equal.
- E.g., **(s6, NULL, NULL, 2000)** indicates that the **Name** and **Major** components of this tuple are missing or unknown. If both are missing, they obviously need not represent the same value!

# Keys: definition

- A subset of attributes of the schema of a relation is a **key** if we do not allow two different tuples in any of its possible relation instances to have the same values across the attributes of the key.
- A key of a relation is a **constraint** of that relation.

# Keys: example

- The key attributes of each relation are underlined.

Student

Sid	Sname	Major	Byear
s1	John	CS	1990
s2	Ellen	Math	1995
s3	Eric	CS	1990
s4	Ann	Biology	2001

Course

Cno	Cname	Dept
c1	Dbs	CS
c2	Calc1	Math
c3	Calc2	Math
c4	AI	Info

Enroll

Sid	Cno	Grade
s1	c1	B
s1	c2	A
s2	c3	B
s3	c1	A
s3	c2	C

# Keys: example

- The key attributes of each relation are underlined.

Student

<u>Sid</u>	Sname	Major	Byear
s1	John	CS	1990
s2	Ellen	Math	1995
s3	Eric	CS	1990
s4	Ann	Biology	2001

Course

<u>Cno</u>	Cname	Dept
c1	Dbs	CS
c2	Calc1	Math
c3	Calc2	Math
c4	AI	Info

Enroll

<u>Sid</u>	<u>Cno</u>	Grade
s1	c1	B
s1	c2	A
s2	c3	B
s3	c1	A
s3	c2	C

# Keys: example (continued)

- Major is **not** a key for Student, since two different tuples share the value “CS” for this attribute.

Student

<u>Sid</u>	Sname	Major	Byear
s1	John	CS	1990
s2	Ellen	Math	1995
s3	Eric	CS	1990
s4	Ann	Biology	2001

# Keys: example (continued)

- Sid alone is **not** a key for Enroll. Also, Cno alone is **not** a key for Enroll.

Enroll

<u>Sid</u>	<u>Cno</u>	Grade
s1	c1	B
s1	c2	A
s2	c3	B
s3	c1	A
s3	c2	C

# Keys: significance

- Each key value of a tuple acts as a **unique name (reference, identifier)** for that tuple.
- Consequently, when a key value appears somewhere in the database, we can equivalently think of the appearance of the referenced tuple at that place in the database.

Enroll

	<u>Sid</u>	<u>Cno</u>	Grade
$s_1, c_1 \rightarrow$	s1	c1	B
$s_1, c_2 \rightarrow$	s1	c2	A
$s_2, c_3 \rightarrow$	s2	c3	B
$s_3, c_1 \rightarrow$	s3	c1	A
$s_3, c_2 \rightarrow$	s3	c2	C

# Keys: significance (continued)

Consequently, when a key value appears somewhere in the database, we can equivalently think of the appearance of the referenced tuple at that place in the database.

Enroll

Sid	Cno	Grade
s1 (John, CS, 1990)	c1(Dbs, CS)	B
s1 (John, CS, 1990)	c2(Calc1, Math)	A
...		

# Keys: significance (continued)

- In contrast, a non-key value references a **set** of tuples; each tuple in this set can of course be identified by its key value.
- Example: the Major value “CS” references the set of tuples identified by the key values “s1” and “s3”.

Student

<u>Sid</u>	Sname	Major	Byear
s1	John	CS	1990
s2	Ellen	Math	1995
s3	Eric	CS	1990
s4	Ann	Biology	2001

# The language SQL

- **SQL** is a language to handle relations in the relational model. We distinguish:
  - The SQL **Data Definition Language** (DDL), which is used to define database and relation **schemas**;
  - The SQL **Data Manipulation Language** (DML), which is used to define, modify, and query relation **instances**.

# SQL DDL: defining database schemas

- We define the database schema simply by specifying the statement:

```
CREATE DATABASE University;
```

```
\connect University  
Other SQL statements  
\q
```

# SQL DDL: defining relation schemas

- We define the relation schemas of the three relations in our running example:

Student(Sid, Sname, Major, Byear)   Course(Cno, Cname, Dept)      Enroll(Sid, Cno, Grade)

```
CREATE TABLE Student  
(Sid    TEXT,  
Sname  VARCHAR(30),  
Major   VARCHAR(15),  
Byear   INTEGER,  
PRIMARY KEY (Sid)  
);
```

```
CREATE TABLE Course  
(Cno    TEXT,  
Cname  VARCHAR(20),  
Dept   VARCHAR(15),  
PRIMARY KEY (Cno)  
);
```

```
CREATE TABLE Enroll  
(Sid    TEXT,  
Cno    TEXT,  
Grade  VARCHAR(2),  
PRIMARY KEY (Sid, Cno)  
);
```

# SQL DDL: NOT NULL constraints

- In SQL, it is possible to disallow null values for certain attributes.
- For obvious reasons, it is advisable to disallow null values for key attributes.
- Example:

```
CREATE TABLE Student  
  (Sid      TEXT NOT NULL,  
   Sname    VARCHAR(30),  
   Major    VARCHAR(15),  
   Byear   INTEGER,  
   PRIMARY KEY (Sid)  
 );
```

# SQL DDL: other operations

- The SQL statement **DROP DATABASE** is used to remove a **database**.
- E.g., **DROP DATABASE University** removes the database **University**
- The SQL statement **DROP TABLE** is used to **remove** a **relation**.
- E.g., **DROP TABLE Student** removes the relation **Student**.

HI, THIS IS  
YOUR SON'S SCHOOL.  
WE'RE HAVING SOME  
COMPUTER TROUBLE.



OH, DEAR - DID HE  
BREAK SOMETHING?

IN A WAY - )



DID YOU REALLY  
NAME YOUR SON  
Robert'); DROP  
TABLE Students;-- ?

OH, YES. LITTLE  
BOBBY TABLES,  
WE CALL HIM.

WELL, WE'VE LOST THIS  
YEAR'S STUDENT RECORDS.  
I HOPE YOU'RE HAPPY.



AND I HOPE  
YOU'VE LEARNED  
TO SANITIZE YOUR  
DATABASE INPUTS.

# SQL DML: defining relation instances

- The SQL statement **INSERT INTO** is used to populate relations.
- Example: `INSERT INTO Student VALUES('s1', 'John', 'CS', 1990);  
INSERT INTO Student VALUES('s2', 'Ellen', 'Math', 1995);  
INSERT INTO Student VALUES('s3', 'Eric', 'CS', 1990);  
INSERT INTO Student VALUES('s4', 'Ann', 'Biology', 2001);`

creates

Student

<u>Sid</u>	<u>Sname</u>	<u>Major</u>	<u>Byear</u>
s1	John	CS	1990
s2	Ellen	Math	1995
s3	Eric	CS	1990
s4	Ann	Biology	2001

# SQL DML: defining relation instances (alternative)

- The SQL statement **INSERT INTO** is used to populate relations.
- Alternative:: `INSERT INTO Student VALUES('s1', 'John', 'CS', 1990),  
('s2', 'Ellen', 'Math', 1995),  
('s3', 'Eric', 'CS', 1990),  
('s4', 'Ann', 'Biology', 2001);`

also creates

Student

<u>Sid</u>	Sname	Major	Byear
s1	John	CS	1990
s2	Ellen	Math	1995
s3	Eric	CS	1990
s4	Ann	Biology	2001

- However, a fifth tuple insertion

```
INSERT INTO Student VALUES ('s1', 'Linda', 'Math', 1993);
```

will be rejected, because ('s1', 'John', 'CS', 1990) is already in the Student relation, and therefore the new insertion would violate the primary key constraint on Sid.

# SQL DML: retrieving relation instances

- To retrieve a relation instance, use the SQL statement **SELECT FROM**.
- Example: `SELECT * FROM Student;`  
`SELECT * FROM Course;`  
`SELECT * FROM Enroll;`

retrieves the relation instances of our running example.

# SQL DML: querying

- A single relation can be queried by selectively retrieving parts of that relation based on a condition.  
**(SELECT FROM WHERE)**

- Example: 

```
SELECT S.sid, S.sname  
FROM Student S  
WHERE S.major = 'CS';
```

retrieves sid and name of each student majoring in CS:

Sid	Sname
s1	John
s3	Eric

# SQL DML: querying (continued)

- Multiple relations can be queried by linking (**joining**) them.
- Example: 

```
SELECT S.sid, S.sname, E.cno
FROM Student S, Enroll E
WHERE S.sid = E.sid AND E.grade = 'B';
```

returns each (sid, sname, cno) tuple where student sid received a B in course cno.

Sid	Cno
s1	c1
s2	c3

# SQL DML: querying (continued)

- Set operations (**UNION**, **INTERSECT**, **EXCEPT**) can be applied to multiple SQL queries
- Example:

```
(SELECT S.sid FROM Student S)
EXCEPT
(SELECT E.sid FROM Enroll E WHERE E.Grade = 'A');
```

selects the sid values of students who did not receive a 'A' grade in any of their courses

Sid
s2
s4

# Referential integrity

- We already encountered primary keys, which impose constraints on single relations.
- Other constraints involve **multiple** relations: in the relation **Enroll**, the tuple **(s<sub>1</sub>, c<sub>1</sub>, 'B')** only makes sense because **s<sub>1</sub>** references the student identified by this sid in the relation **Student** and **c<sub>1</sub>** references the course identified by this cno.
- However, inserting the tuple **(s<sub>5</sub>, c<sub>1</sub>, 'A')** into **Enroll** makes no sense, since s<sub>5</sub> does not reference any student in **Student**.

# Foreign keys

- In the relation schemas

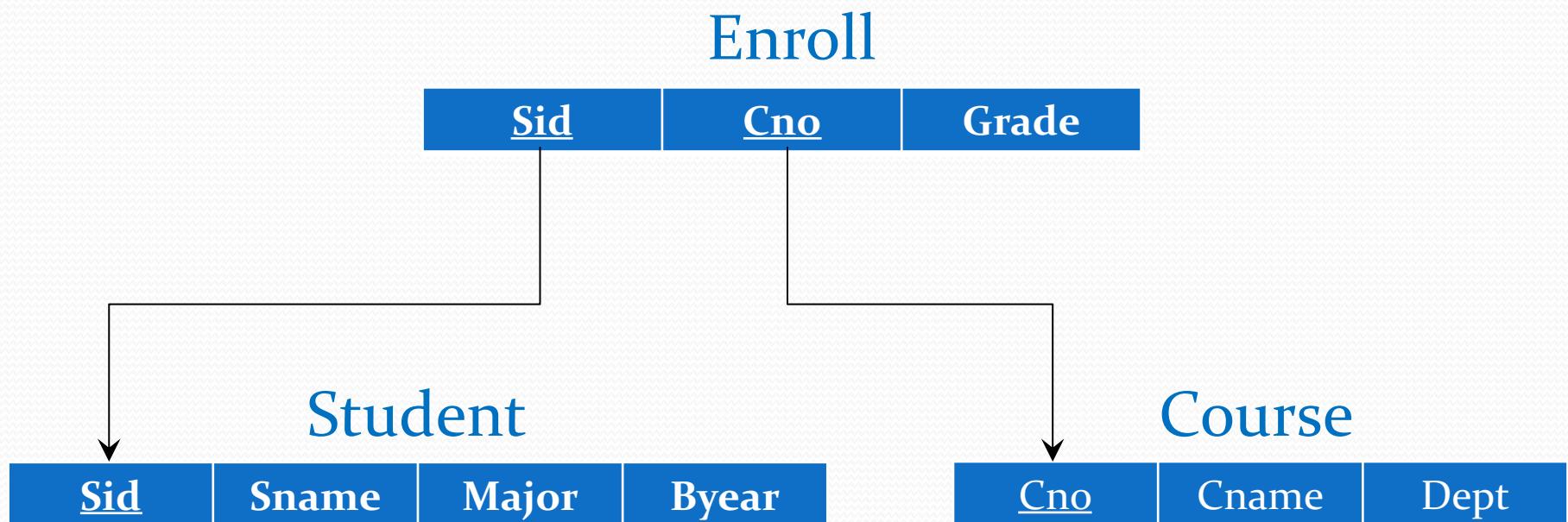
Student(Sid, Sname, Major, Byear)

Course(Cno, Cname, Dept)

Enroll(Sid, Cno, Grade)

the attribute **Sid** of **Enroll** is a **foreign key** referencing the **primary key** **Sid** of **Student**. Likewise, the attribute **Cno** of **Enroll** is a **foreign key** referencing the **primary key** **Cno** of **Course**.

- Graphically:



# Foreign keys in SQL

- The foreign key constraints to Enroll can be added as follows:

```
CREATE TABLE Enroll
  (Sid    INTEGER,
   Cno    INTEGER
   Grade VARCHAR(2),
   PRIMARY KEY (Sid, Cno),
   FOREIGN KEY (Sid) REFERENCES Student(SID),
   FOREIGN KEY (Cno) REFERENCES Course(Cno)
);
```

# Foreign keys and insertion

- Foreign keys impose an order on insertions in relations: the tuple  $(s_1, c_1, 'B')$  can only be inserted in **Enroll** **after** a tuple identified by the **Sid** value  $s_1$  exists in **Student** **and after** a tuple identified by the **Cno** value  $c_1$  exists in **Course**.
- Hence, inserting the tuple  $(s_5, c_1, 'A')$  in the current state of **Enroll** will be rejected, since there is no tuple in **Student** with **Sid** value  $s_5$ .

# Foreign keys and deletion

- Foreign keys also necessitate **cascading deletions**: deleting the tuple `(s1, 'John', 'CS', 1990)` from `Student` requires deleting all tuples in `Enroll` referencing the `Sid` value `s1`. (A similar effect occurs upon deleting a tuple from `Course`.)

# Foreign keys and deletion (c'ed)

- Two option in SQL:
  - Allow cascading deletions.
  - Disallow cascading deletions (and hence refuse deletion).
- Example:

```
CREATE TABLE Enroll
  (Sid    INTEGER,
   Cno    INTEGER
   Grade VARCHAR(2),
   PRIMARY KEY (Sid, Cno),
   FOREIGN KEY (Sid) REFERENCES Student(SID) ON DELETE CASCADE,
   FOREIGN KEY (Cno) REFERENCES Courses(Cno) ON DELETE RESTRICT
 );
```

# SQL – Part 1

# Running example: assumptions

- We use the following relation schemas:
  - `Student(Sid, Sname, Major, Byear)`
  - `Course(Cno, Cname, Dept)`
  - `Enroll(Sid, Cno, Grade)`
- `Sid` and `Cno` are foreign keys in `Enroll` referencing `Sid` and `Cno` in `Student` and `Course`, respectively.
- We will also assume that `Student` has  $m$  tuples, `Enroll` has  $n$  tuples, and `Course` has  $k$  tuples.

# SQL queries – basic form

- The simplest form of a SQL query is as follows:

```
SELECT list of components of tuple variables
FROM   list of tuple variables associated with relations
WHERE  condition on components of tuple variables;
```

# Example of a simple SQL query

- Consider the query “Find the sids and names of students who major in CS.”
- In SQL, the formulation is,

```
SELECT S.Sid, S.Sname  
FROM Student S  
WHERE S.Major = 'CS';
```

# Semantics of a basic SQL query

- In the SQL query    `SELECT S.Sid, S.Sname  
FROM Student S  
WHERE S.Major = 'CS';`

`S` is a tuple variable that ranges over all tuples in the `Student` relation. So, if this relation has 10 tuples, then `S` ranges over these 10 tuples.

- The semantics of this query is

for each tuple `S` in `Student`  
    if `S.Major = 'CS'`  
        then output (`S.Sid, S.Sname`)

# Semantics (continued)

- It is important to realize that, when a tuple variable is associated with a relation, then that tuple variable is assigned, one-at-a-time, to each tuple in that relation.
- E.g., if **Student** is

<u>Sid</u>	Sname	Major	Byear	
s1	John	CS	1990	$\leftarrow S$
s2	Ellen	Math	1995	$\leftarrow S$
s3	Eric	CS	1990	$\leftarrow S$
s4	Ann	Biology	2001	$\leftarrow S$

then **S** is assigned, one at the time, to these four tuples.

- The order in which **S** is assigned to these tuples is not pre-determined since the **Student** relation is a set!

# Semantics (continued)

- For each tuple assigned to the variable **S**, the (boolean) condition in the **WHERE** clause is checked.
- If the condition is true for that tuple, then the attributes of that tuple specified in the **SELECT** clause will be output.
- If the condition is false for that tuple, no attributes of that tuple are output.

# SQL has bag semantics

- Consider the query “Find the majors of students.”
- In SQL:

```
SELECT S.Major  
FROM Student S
```
- Semantics: **for each tuple S in Student  
output (S.Major)**
- Output on our sample database:
- Observe that ‘CS’ appears twice: the result of this query is a **bag (multiset)**

Major
CS
Math
CS
Biology

# Enforcing set semantics

- To coerce the result of the query into a set, use the **DISTINCT** clause:

```
SELECT DISTINCT S.Major  
FROM Student S
```

- The result is the set

Major
CS
Math
Biology

# Time-complexity of DISTINCT

- The time-complexity of the coercion from the bag into the set is at least linear in the size of the bag.
- If **hashing** is used, it can be done during the output generation.
- One can also use **sorting** as a technique to accomplish the coercion. In this case, the time-complexity is  $O(n \log n)$ , where  $n$  is the number of tuples in the bag.

# EXPLAIN in PostgreSQL

- In PostgreSQL, it is possible to see how the system implements the coercion.
- To see this, issue the **EXPLAIN** command:

```
EXPLAIN SELECT DISTINCT S.Major  
          FROM Student S;
```

- The system will generate an **access plan** that indicates whether hashing or sorting is used.
- In general, **EXPLAIN** is very useful to understand how an SQL query is processed by the system.

# EXPLAIN ANALYZE in PostgreSQL

- In PostgreSQL, it is also possible to see prediction and actual time and space complexity associated with a query.
- To see this, issue the **EXPLAIN ANALYZE** command:

```
EXPLAIN ANALYZE SELECT DISTINCT S.Major  
FROM Student S;
```

# Renaming attributes

- In SQL, it is possible to give names to attributes that appear in a SELECT clause using the **AS** clause:

```
SELECT S.Sid AS Identifier, S.Sname AS Name  
FROM Student S  
WHERE S.Major = 'CS';
```

- This query returns the relation

Identifier	Name
s1	John
s3	Eric

# Ordering the output

- In SQL, it is possible to sort the output on the basis of order relation that exists on attribute domains
- This can be done with the **ORDER BY** clause

```
SELECT S.sid, S.sname  
FROM Student S  
WHERE S.Major = 'CS'  
ORDER BY sname;
```

sid	sname
s3	Eric
s1	John

# Ordering the output (by column)

- The following queries are equivalent

```
SELECT S.sid, S.sname  
FROM Student S  
WHERE S.Major = 'CS'  
ORDER BY sname;
```

```
SELECT S.sid, S.sname  
FROM Student S  
WHERE S.Major = 'CS'  
ORDER BY 2;
```

# Randomly ordering the output

- In SQL, it is possible to randomly order the output of a query
- This can be done with the **ORDER BY RANDOM()** clause
- The following executions of the same queries may not give the same result

```
SELECT *  
FROM Student S  
ORDER BY RANDOM();
```

```
SELECT *  
FROM Student S  
ORDER BY RANDOM();
```

# Queries involving multiple relations

- Consider the query “Find the names of students along with the cnos of the courses in which they received a B grade”:

```
SELECT S.Sname, E.Cno  
FROM Student S, Enroll E  
WHERE S.Sid = E.Sid AND E.Grade = 'B';
```

- We have **two** tuple variables:
  - S** ranges over the tuples in **Student**;
  - E** ranges over the tuples in **Enroll**.
- In the **WHERE** clause, conditions are checked on attribute values of tuples assigned to these variables.

# Queries on multiple relations (c'ed)

- SQL query:

```
SELECT S.Sname, E.Cno  
FROM Student S, Enroll E  
WHERE S.Sid = E.Sid AND E.Grade = 'B';
```

- Semantics:

for each tuple  $S$  in Student  
    for each tuple  $E$  in Enroll  
        if  $S.Sid = E.Sid$  and  $E.Grade = 'B'$   
            then output  $(S.Sname, E.Cno)$

- Notice the double-nested for-loop!

# Queries on multiple relations (c'ed)

- SQL query:

```
SELECT S.Sname, C.Cno  
FROM Student S, Enroll E  
WHERE S.Sid = E.Sid AND E.Grade = 'B'
```

- The condition **S.Sid = E.Sid** is called a **join condition**: it verifies a condition between two variables
- The condition **E.Grade = 'B'** is called a **constant comparison** condition: it verifies a condition on a single variable.

# Time-complexity considerations

- If Student has  $m$  tuples and Enroll has  $n$  tuples, then the if statement  
 $\text{if S.Sid} = \text{E.Sid and E.Grade} = \text{'B'}$   
is evaluated  $m \times n$  times.
- This essentially quadratic time complexity compromises the scalability of this simple query for large values of  $m$  and  $n$ .

# Sub-queries in the FROM clause

- The previous query can also be formulated as follows:

```
SELECT S.Sname, C.Cno  
FROM Student S, (SELECT E.Sid, E.Cno  
                  FROM Enroll E  
                  WHERE E.Grade = 'B') C  
WHERE S.Sid = C.Sid;
```

- There are 2 **global** variables (**S, C**) and 1 **local** variable (**E**):
  - **S** ranges over the tuples in **Student**;
  - **C** ranges over the tuples in the relation determined by the **sub-query** (the part between the brackets);
  - **E** ranges over the tuples in **Enroll**.

# Subqueries (continued)

- We conclude the following:
  - In SQL, it is permissible to have tuple variables that range over relations, but also tuple variables that range over the tuples of a relation computed by a subquery.
  - This works, because an SQL query always returns a relation.
  - Subqueries are allowed to occur in the **FROM** clause of a query or another subquery.

# Performance improvement

- Query structure:  

```
SELECT S.Sname, C.Cno
FROM Student S, (subquery) C
WHERE S.Sid = C.Sid
```
- Semantics:  

```
for each tuple S in Student
    for each tuple C in output (subquery)
        if S.Sid = C.Sid
            then output (S.Sname, C.Cno)
```
- If the subquery yields  $s$  tuples, then the if-then statement will be executed  $m \times s$  times instead of  $m \times n$ .
- This can be a considerable gain if  $s$  is small compared to  $m$ .

- Of course we must also add the time to compute the relation generated by the subquery. This can be done using a linear scan requiring  $n$  operations.
- So the total time is  $n + m \times s$  instead of  $m \times n$ .

# Multiple tuple variables ranging over the same relation

- Consider the query “Find the name of each student enrolled in at least **two** courses.”
- Since a tuple variable ranging over a relation is only assigned to a single tuple of that relation, it can never hold information about two or more tuples.
- However, if we have two or more tuple variables ranging over the same relation, then this becomes possible.

# Multiple tuple variables ranging over the same relation: example

- We want to express the query “Find the sid and name of each student enrolled in at least **two** courses.”
- In SQL, this query can be formulated using two variables ranging over the **Enroll** relation:

```
SELECT DISTINCT S.Sid, S.Sname  
FROM Student S, Enroll E1, Enroll E2  
WHERE S.Sid = E1.Sid AND S.Sid = E2.Sid AND E1.Cno <> E2.Cno;
```

- $\text{<>}$  is the symbol for not equal, i.e.,  $\neq$ .
- Why do we need DISTINCT?

# Multiple tuple variables ranging over the same relation: example (continued)

- Semantics:
  - for each tuple S in Student
  - for each tuple E1 in Enroll
  - for each tuple E2 in Enroll
    - if S.Sid = E1.Sid and S.Sid = E2.Sid and E1.Cno ≠ E2.Cno
    - then output (S.Sid,S.Sname)
- Performance:  $m \times n^2$ .
- Later, we shall see that there are far more efficient ways to write and solve this query.

# Queries using set operations

- We now consider queries involving the set operations **union**, **intersection**, and **difference**:

Operation	Math notation	SQL notation
Union	$\cup$	UNION
Intersection	$\cap$	INTERSECT
Difference	$-$	EXCEPT

# Set operations: examples

*R*

A	B
1	1
1	2
3	2
3	1

*S*

A	B
1	3
3	1
3	2
2	2
2	3

$R \cup S$

A	B
1	1
1	2
3	2
3	1
1	3
2	2
2	3

$R \cap S$

A	B
3	2
3	1

$R - S$

A	B
1	1
1	2

$S - R$

A	B
1	3
2	2
2	3

Notice that  $R \cup S = S \cup R$  and  $R \cap S = S \cap R$ .  
 However, in general,  $R - S \neq S - R$ .

# Union

- Consider the query “Find the sids and names of all students who major in CS **or** who major in Math.”
- In SQL,

```
(SELECT S.Sid, S.Sname  
FROM Student S  
WHERE S.Major = 'CS')  
UNION  
(SELECT S.Sid, S.Sname  
FROM Student S  
WHERE S.Major = 'Math');
```

# Union (alternative)

- Consider again “Find the sids and names of all students who major in CS **or** who major in Math.”
- In SQL,

```
SELECT S.Sid, S.Sname  
FROM Student S  
WHERE S.Major = 'CS' OR S.Major = 'Math';
```

# Intersection

- Consider the query “Find the sids of all students who are enrolled in course c1 **and** course c2.”
- In SQL,

```
(SELECT E.Sid  
FROM Enroll E  
WHERE E.Cno = 'c1')  
INTERSECT  
(SELECT E.Sid  
FROM Enroll E  
WHERE E.Cno = 'c2');
```

# Wrong alternative for intersection

- Consider again “Find the sids of all students who major in  $c_1$  **and**  $c_2$ .”
- What about the following SQL query?

```
SELECT E.Sid  
FROM Enroll E  
WHERE E.Cno = 'c1' AND E.Cno = 'c2';
```

- The result of this query is the **empty set** since, for no tuple from **Enroll** assigned to the variable **E**, **E.Cno** can be equal to both **c1** and **c2** at the same time!

# Difference

- Consider the query “Find the sids of all students who are enrolled in **no** courses.”
- In SQL,  

```
(SELECT S.Sid  
  FROM Student S)  
EXCEPT  
(SELECT E.Sid  
  FROM Enroll E);
```
- The first subquery yields the set of all student sids.
- The second subquery yields the set of the sids of all students enrolled in some course.
- The difference between both sets yields the result.

# Performance considerations

- UNION, INTERSECTION, and DIFFERENCE can be implemented using hashing or using sorting.
- Therefore, these operations can be implemented in linear time  $O(m + n)$  or in  $O(n \log n + m \log m)$ , where  $m$  and  $n$  are the respective sizes of the relations involved in these operations.

# Semantics of set operations in SQL

- In SQL, the result of the **UNION**, **INTERSECT**, and **EXCEPT** operations are **sets**, even if the inputs to these operations are bags.

- So, the result of

```
(SELECT E.Sid  
  FROM Enroll E)  
UNION  
(SELECT E.Sid  
  FROM Enroll E)
```

is the same as the result of 

```
SELECT DISTINCT E.Sid  
  FROM Enroll E
```

- To retain **bag** semantics, use **UNION ALL**, **INTERSECT ALL**, and **EXCEPT ALL**.

# SQL – Part 2

Sub-queries and Set Predicates

# The relational model

- Our running example.

Student				Course		
Sid	Sname	Major	Byear	Cno	Cname	Dept
s1	John	CS	1990	c1	Dbs	CS
s2	Ellen	Math	1995	c2	Calc1	Math
s3	Eric	CS	1990	c3	Calc2	Math
s4	Ann	Biology	2001	c4	AI	Info

Enroll

Sid	Cno	Grade
s1	c1	B
s1	c2	A
s2	c3	B
s3	c1	A
s3	c2	C

# The IN and NOT IN predicates

- Consider the query “Find the sids of students who are enrolled in CS courses.”
- In SQL with join conditions:

```
SELECT DISTINCT E.Sid  
FROM Enroll E, Course C  
WHERE E.Cno = C.Cno AND C.Dept = 'CS';
```

# The IN predicate

- “Find the sids of students who are enrolled in CS courses.”
- In SQL with the **IN** predicate,

```
SELECT DISTINCT E.Sid  
FROM Enroll E  
WHERE E.Cno IN (SELECT C.Cno  
                  FROM Course C  
                  WHERE C.Dept = 'CS');
```

- The **IN** predicate corresponds to the set-membership predicate  $\in$ .
- For example,  $a \in \{a, b, c\}$  is true, but  $d \in \{a, b, c\}$  is false.

# The NOT IN predicate

- Now consider the SQL query

```
SELECT DISTINCT E.Sid  
FROM Enroll E  
WHERE E.Cno NOT IN (SELECT C.Cno  
                     FROM Course C  
                     WHERE C.Dept = 'CS');
```

- This is the query “Find the sids of students who are enrolled in a course that is not offered by the CS department.”
- Note that this is not the query “Find the sids of students who take no CS courses.”

# The SOME predicate

- Consider again the query “Find the sids of students who are enrolled in a CS course”:

```
SELECT DISTINCT E.Sid  
FROM Enroll E  
WHERE E.Cno IN (SELECT C.Cno FROM Course C WHERE C.Dept = 'CS');
```

- This query can also be written using the **SOME** predicate as follows:

```
SELECT DISTINCT E.Sid  
FROM Enroll E  
WHERE E.Cno = SOME (SELECT C.Cno FROM Course C WHERE C.Dept = 'CS');
```

- The **= SOME** predicate checks if **E.Cno** is equal to some (i.e., at least one) number of a CS course.

# The ALL predicate

- Consider the relation

Pid	Age
p1	10
p2	9
p3	12
p4	9

- Consider the query “Find the pids of the youngest persons.”
- The answer to this query consists of p2 and p4 since their age (i.e., 9) is smaller than or equal to all ages, i.e., (10, 9, 12, 9):

Pid
p2
p4

- In SQL with the **ALL** predicate,

```
SELECT P.Pid  
FROM Person P  
WHERE P.Age <= ALL (SELECT P1.Age FROM Person P1);
```

# More SOME and ALL predicates

- Consider the SQL query

```
SELECT P.Pid  
FROM Person P  
WHERE P.Age < ALL (SELECT P1.Age  
                     FROM Person P1);
```

- This query will return nothing since there is no person whose age is strictly smaller than all ages.
- What are the answers of similar queries with the following predicates?

SOME	ALL
= SOME	= ALL
<> SOME	<> ALL
< SOME	< ALL
<= SOME	<= ALL
> SOME	> ALL
>= SOME	>= ALL

# The EXISTS predicate

- The **EXISTS** predicate takes as argument a relation that is the answer of a query.
- If there exists a tuple in that relation, then the **EXISTS** predicate evaluates to **true**.
- Otherwise, if there does not exist a tuple in that relation, then the **EXISTS** predicate evaluates to **false**.

# The EXISTS predicate: example

- Assume that there exist students who major in CS, then  
`EXISTS (SELECT S.Sid  
 FROM Student S  
 WHERE S.Major = 'CS')`  
evaluates to **true**.
- If there are no students who major in biology, then  
`EXISTS (SELECT S.Sid  
 FROM Student S  
 WHERE S.Major = 'Biology')`  
evaluates to **false**.

# The NOT EXISTS predicate

- The NOT EXISTS predicate takes as argument a relation that is the answer of a query.
- If there exists a tuple in that relation, then the NOT EXISTS predicate evaluates to **false**.
- Otherwise, i.e., if there does not exist a tuple in that relation, then the NOT EXISTS predicate evaluates to **true**.

# The NOT EXISTS predicate: example

- Assume that there exist students who major in CS, then  
`NOT EXISTS (SELECT S.Sid  
FROM Student S  
WHERE S.Major = 'CS')`  
evaluates to **false**.
- If there are no students who major in biology, then  
`NOT EXISTS (SELECT S.Sid  
FROM Student S  
WHERE S.Major = 'Biology')`  
evaluates to **true**.

# Emptiness check of a relation

- The **EXISTS** predicate determines whether or not a relation is **not empty**.
- The **NOT EXISTS** predicates determines whether or not a relation is **empty**.

Predicate	Relation	Value
EXISTS	$\neq \emptyset$	true
EXISTS	$= \emptyset$	false
NOT EXISTS	$\neq \emptyset$	false
NOT EXISTS	$= \emptyset$	true

## EXISTS and NOT EXISTS in WHERE clause

- Since **EXISTS** and **NOT EXISTS** are boolean predicates, they can be used in the **WHERE** clause of a SQL query.
  - Example: “Find the sids of students whose name is John provided that **there exist** students who major in CS.”

- This query will return the sids of all students with name John, but only if there are student majoring in CS.
- If, however, there are no such students, then the result of this query is the empty relation.

# Sub-queries with parameters

- The power of the **EXISTS** and **NOT EXISTS** predicates really emerges when the argument query has **parameters**.
- Consider the query “Find the sids of student who are enrolled in a course.”
- In SQL with the **EXISTS** predicate,

```
SELECT S.Sid  
FROM Student S  
WHERE EXISTS (SELECT E.Cno  
               FROM Enroll E  
               WHERE S.Sid = E.Sid);
```

- Notice how the Student tuple variable **S** is a parameter of the inner sub-query.

# Global and local variables in SQL

- Consider `EXISTS (SELECT E.Cno  
FROM Enroll E  
WHERE S.Sid = E.Sid)`
- In this predicate, `S` is a **global** variable, whereas `E` is a **local** variable.
- The possible values for the parameter `S.Sid` are coming from the outside: for each tuple `S` in the `Student` relation, `S.Sid` takes on the sid value of that tuple.

- Now, if the value of `S.Sid` is the sid of a student who takes a course, then

```
EXISTS (SELECT E.Cno  
        FROM Enroll E  
        WHERE S.Sid = E.Sid)
```

evaluates to **true** and, therefore, the sid of such a student is returned by the (outer) query.

- If, however, the value of **S.Sid** is the sid of a student who does **not** takes any course, then

```
EXISTS (SELECT E.Cno  
        FROM Enroll E  
        WHERE S.Sid = E.Sid)
```

evaluates to **false** and, therefore, this sid value is **not** returned by the outer query.

# Global and local variables: example

- We now want to find the sids of students who do **not** take any courses.
- For that purpose, we can write the following query:

```
SELECT S.Sid  
FROM Student S  
WHERE NOT EXISTS (SELECT E.Cno  
                   FROM Enroll E  
                   WHERE S.Sid = E.Sid);
```

# Putting it all together

- Fairly complex queries can be composed.
  - Example: find the majors of students named Ellen who do not take any CS course.

- Next consider the query  
“Find the sids of students who take **all** CS courses.”
- This query can be reformulated as follows:  
“Find the sids of students for whom there does **not exist** a CS course they are **not** enrolled **in**.”
- The **not exist** and **not in** the latter statement should suggest a way to write this query in SQL.

- This SQL query is as follows:

```
SELECT S.Sid  
FROM Student S  
WHERE NOT EXISTS (SELECT C.Cno  
                   FROM Course C  
                   WHERE C.Dept = 'CS' AND  
                         C.Cno NOT IN (SELECT E.Cno  
                           FROM Enroll E  
                           WHERE S.Sid = E.Sid))
```

- Notice that a parameter like S.Sid can occur inside a (NOT) EXISTS as well as inside a (NOT) IN predicate.

# Views

Views, materialized views, temporary views, parameterized views, and recursive views

# The database

- A **relational database** is a collection of **base relations**

Student

Sid	Sname	Major	Byear
s1	John	CS	1990
s2	Ellen	Math	1995
s3	Eric	CS	1990
s4	Ann	Biology	2001

Course

Cno	Cname	Dept
c1	Dbs	CS
c2	Calc1	Math
c3	Calc2	Math
c4	AI	CS

Enroll

Sid	Cno	Grade
s1	c1	B
s1	c2	A
s2	c3	B
s3	c4	A
s3	c2	C

# View

- CS view:

**CS\_Course**

Cno	Cname
c1	Dbs
c4	AI

**Student\_enrolled\_in\_CS\_course**

Sid	Sname	Major	Byear
s1	John	CS	1990
s3	Eric	CS	1990

**CS\_Enroll\_Info**

Sid	Cno	Grade
s1	c1	B
s1	c2	A
s3	c2	C
s3	c4	A

# Views

- A **view** is a virtual relation defined by a query
- Example view in SQL:

```
CREATE VIEW CS_Course AS  
    SELECT C.Cno, C.Cname  
    FROM Course C  
    WHERE C.Dept = 'CS';
```

- This view augments the database with a new relation that contain the (Cno,Cname) pairs for each course offered by the CS department

# Views

- A **view** is a virtual relation defined by a query

```
CREATE VIEW CS_Course AS  
    SELECT C.Cno, C.Cname  
    FROM Course C  
    WHERE C.Dept = 'CS';
```



CS\_Course

Cno	Cname
c1	Dbs
c4	AI

# Database = Base relations + virtual relations (views)

Student

Sid	Sname	Major	Byear
s1	John	CS	1990
s2	Ellen	Math	1995
s3	Eric	CS	1990
s4	Ann	Biology	2001

Course

Cno	Cname	Dept
c1	Dbs	CS
c2	Calc1	Math
c3	Calc2	Math
c4	AI	CS

Enroll

Sid	Cno	Grade
s1	c1	B
s1	c2	A
s2	c3	B
s3	c1	A
s3	c2	C

CS\_course

Cno	Dbs
c1	Dbs
c4	AI

# Views (terminology)

```
CREATE VIEW CS_Course AS  
    SELECT C.Cno, C.Cname  
    FROM Course C  
    WHERE C.Dept = 'CS';
```

- CS\_Course is the **name** of the view (i.e., the name of the virtual relation).
- The SELECT query is the **definition** of the view.

# Views in queries

- Once defined, views can be used as relations in queries.
- Example:

```
SELECT C.Cname  
FROM CS_Course C;
```

returns the names of the courses offered by the CS department.

# Views in queries: example

- Consider the query “Find the names of students enrolled in CS courses.”
- In SQL,

```
SELECT DISTINCT S.Sname  
FROM Student S, Enroll E, CS_Course C  
WHERE S.Sid = E.Sid AND E.Cno = C.Cno;
```

[view relation](#)

# Views defined using other views

- Views may also be used in the definition of other views.
  - Example: create the view showing all information of students taking CS courses.
  - In SQL,

# Views defined using other views

- Example: create a view showing all enrollment information of students who take CS courses
  - In SQL,

# Importance of views

- Views can be used to **modularize** a complex query into simpler units. The previous examples illustrate this.
- Views provide **independence** from changes in the underlying database due to **growth** and **restructuring** of the base relation schemas.
- This is called **logical data independence**

# Importance of views: Growth

- Database schemas can grow in several ways:
  - A base table can be expanded by the addition of new attributes and constraints (e.g., using `ALTER`).
  - New base tables or views can be added to the database.
- The definition of an existing view need **not** be changed in these cases!

# Views unaffected by growth

- Addition of a new columns to a base relation does not affect any views defined using this base relation

```
ALTER TABLE Student  
    ADD COLUMN Matriculation_Year INTEGER;
```

- The CS\_course and Student\_enrolled\_in\_CS\_course views do not need to be re-defined because of this alteration to the Student relation

# Importance of views: Restructuring

- It might be necessary to restructure the database in a more fundamental way.
- For example, it could become necessary to replace the relation **Student(Sid, Sname, Major, Byear)** by the two relations

Student\_Info (Sid, Sname, Byear)

Student\_Major(Sid, Major);

- To ensure that previously defined views remain valid, it now suffices to define **Student** as a view using the base relations **Student\_Info** and **Student\_Major**.

# Restructuring (continued)

- This can easily be done as follows:

CREATE VIEW Student AS

```
SELECT S.Sid AS Sid, S.Sname AS Sname, S.Byear AS Byear, Sm.Major AS Major  
FROM Student_Info S, Student_Major Sm  
WHERE S.Sid = Sm.Sid;
```

- Any query or view that previously used the base relation Student does not need to be changed

# Garbage collection

- As discussed, views can be defined in terms of base relations and other views.
- This has consequences on the semantics of the DROP TABLE and DROP VIEW commands;
- DROP TABLE *table\_name* will only succeed if there are no views with *table\_name* as base relation.
- DROP TABLE *table\_name* CASCADE will succeed and drop all the views that were defined in terms of *table\_name*
- The same applies to DROP VIEW *view\_name*

# Review of our discussion on views

- Views are defined by queries.
- Recall our example: 

```
CREATE VIEW CS_Course AS  
    SELECT C.Cno, C.Cname  
    FROM Course C  
    WHERE C.Dept = 'CS';
```

- The following SQL query uses this view:

```
SELECT E.Sid  
FROM Enroll E  
WHERE E.Cno IN (SELECT C.Cno  
    FROM CS_Course C);
```

It returns the sids of students enrolled in CS courses.

# Query evaluation with view expansion

- View expansion refers to the process of rewriting queries by expanding the views by their definitions in these queries

```
CREATE VIEW CS_Course AS  
    SELECT C.Cno, C.Cname  
    FROM Course C  
    WHERE C.Dept = 'CS';
```

```
SELECT C.Cname  
FROM CS_Course C
```

view expansion

```
SELECT C.Cname  
FROM (SELECT C.Cno, C.Cname  
      FROM Course C  
     WHERE C.Dept = 'CS') C;
```

- View expansion precedes query evaluation

# View updates

- Views are essentially designed so that users can work with them as if they were base relations.
- Hence, we do not only want to query views, but also to update views.
- Updating views, however, will necessarily affect the base relations.
- Is it therefore always possible to update views?
- If it is possible, can it be done in a deterministic (algorithmic) way?

# Database = Base relations + virtual relations (views)

Student

Sid	Sname	Major	Byear
s1	John	CS	1990
s2	Ellen	Math	1995
s3	Eric	CS	1990
s4	Ann	Biology	2001

Course

Cno	Cname	Dept
c1	Dbs	CS
c2	Calc1	Math
c3	Calc2	Math
c4	AI	CS

Enroll

Sid	Cno	Grade
s1	c1	B
s1	c2	A
s2	c3	B
s3	c1	A
s3	c2	C

CS\_course

Cno	Dbs
c1	Dbs
c2	AI

# Insertions in views

- Consider again the (very simple) view,

```
CREATE VIEW CS_Course AS  
    SELECT C.Cno, C.Cname  
    FROM Course C  
    WHERE C.Dept = 'CS';
```

- We want to do the following insertion:

```
INSERT INTO CS_Course VALUES(c6, 'Networks');
```

# Insertions in views (continued)

- In SQL, this is “solved” by (internally) executing the following command:

```
INSERT INTO Course VALUES(c6, 'Networks', NULL);
```
- Unfortunately, this will make the `(c6, 'Networks')` disappear from the view again, as the system no longer recognizes this as a `CS` course ...
- It takes quite sophisticated update management to enforce the system to add `(c6, 'Networks', 'CS')` instead.
- In SQL, **triggers** can be used for this purpose.

# After INSERT of (c6, Networks) in CS\_course

Student

Sid	Sname	Major	Byear
s1	John	CS	1990
s2	Ellen	Math	1995
s3	Eric	CS	1990
s4	Ann	Biology	2001

Course

Cno	Cname	Dept
c1	Dbs	CS
c2	Calc1	Math
c3	Calc2	Math
c4	AI	CS
c6	Networks	NULL

Enroll

Sid	Cno	Grade
s1	c1	B
s1	c2	A
s2	c3	B
s3	c1	A
s3	c2	C

CS\_course

Cno	Dbs
c1	Dbs
c2	AI

# Deletions in views

- Deletions in complex views can also be problematic
- Since ‘c1’ appear in the CS\_course view, the following deletion will remove ‘c1’ from the Course relation

```
DELETE FROM CS_course WHERE Cno = ‘c1’;
```

- However, since course ‘c6’ does not appear in the CS\_course view, the following deletion will not remove ‘c6’ from the Course relation

```
DELETE FROM CS_course WHERE Cno = ‘c6’;
```

# After DELETE of course c6 from CS\_course

Student

Sid	Sname	Major	Byear
s1	John	CS	1990
s2	Ellen	Math	1995
s3	Eric	CS	1990
s4	Ann	Biology	2001

Course

Cno	Cname	Dept
c1	Dbs	CS
c2	Calc1	Math
c3	Calc2	Math
c4	AI	CS
c6	Networks	NULL

Enroll

Sid	Cno	Grade
s1	c1	B
s1	c2	A
s2	c3	B
s3	c1	A
s3	c2	C

CS\_course

Cno	Dbs
c1	Dbs
c2	AI

# View materialization

- In general, a view is a virtual relation.
- If it is used in a query, it is computed at the time this query is evaluated.
- This can be very expensive.
- In many cases, especially if the database state is not changing, it is useful to precompute the view and then store it in the database for future use.
- A view that is precomputed is called a **materialized view**.

# Materialized views (challenges)

- The major challenge with materialized views is that they need to be updated when updates to the base relations (i.e., the state of the basis) occur.
- For certain views (like views that are defined by join queries) this can be done efficiently using incremental changes to the materialized views.

# Materialized views (challenges)

- However, when views are defined by queries that involve negation with expressions involving EXCEPT, ALL, NOT IN, and NOT EXISTS, incremental view maintenance may not be possible.
- In such cases, the entire process to materialize the views need to be redone.

# Materialized views (challenges)

```
CREATE VIEW foo AS
  SELECT Cno
    FROM Course
   WHERE NOT EXISTS (SELECT 1
                      FROM Course
                     WHERE cname = 'Java');

  SELECT * FROM foo;
```

Returns all cnos since no course exists with name Java.  
However, after the following insertion, no cnos are returned since a course exists with name Java:

```
INSERT INTO Course VALUES ('c5', 'Java', 'CS');
```

# Materialized views in PostgreSQL

```
CREATE MATERIALIZED VIEW CS_Course AS  
SELECT C.cno  
FROM Course C  
WHERE C.dept = 'CS';
```

... other SQL statements (including INSERT/DELETE)  
These statements may change the state of CS\_course!

```
REFRESH MATERIALIZED VIEW CS_Course;  
After this, we get the correct state for CS_course
```

# Temporary views (WITH statement)

- A common application of views is to use them to modularize the solution for a query
- Such views need not always be defined and retained
- The **WITH** statement of SQL permits us to define **temporary views** which are only local to the query we wish to solve
- These temporary views are not made persistent outside the query

# Temporary views (WITH clause)

- The **WITH** clause provides a way to define **temporary views** for use in a larger query

```
WITH view_1 AS (SQL query 1),  
     view_2 AS (SQL query 2),
```

...

```
view_n AS (SQL query n)
```

```
SQL query that can use view_1,...,view_n;
```

- At the end of the evaluation of the last **SQL query**, the temporary views **view\_1** through **view\_n** are discarded

# Temporary views (example)

“Find the sid and name of each student who takes a Math course and who majors in CS”

```
WITH Math_Course AS (SELECT Cno, Cname
                      FROM Course
                      WHERE Dept = 'Math'),
      Student_enrolled_in_Math_course AS
        (SELECT Sid, Sname, Major, Byear
         FROM Student
         WHERE Sid IN (SELECT E.Sid
                       FROM Enroll E, Math_Course C
                       WHERE E.Cno = C.Cno))
    SELECT Sid, Sname
    FROM Student_enrolled_in_Math_course
    WHERE Major = 'CS';
```

# Recursive views

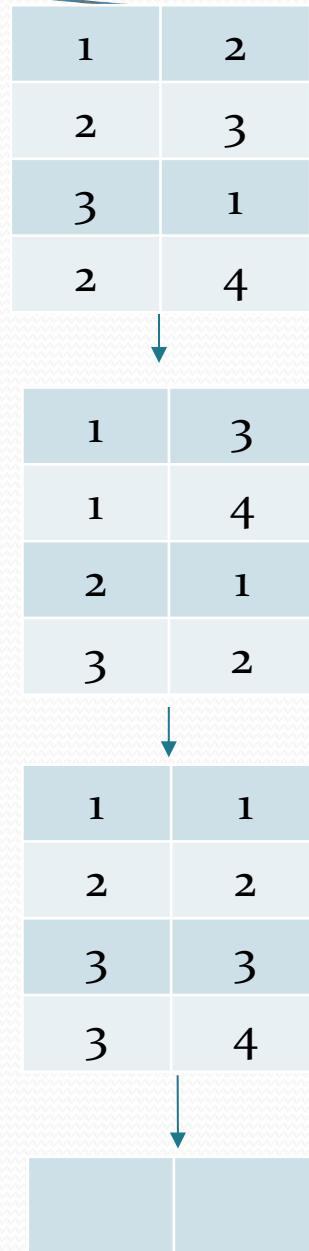
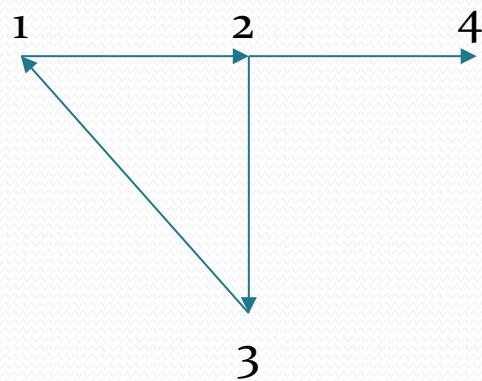
- Consider a directed graph  $\text{Graph}(\text{source}, \text{target})$
- A pair  $(s,t)$  is in  $\text{Graph}$  if  $(s,t)$  is an edge in the graph
- Consider the relation  $\text{Path}(\text{source},\text{target})$
- A pair  $(s,t)$  is in  $\text{Path}$  if there is a path in  $\text{Graph}$  from  $s$  to  $t$
- A recursive definition of  $\text{Path}$  is as follows:

Base rule: If  $\text{Graph}(s,t)$  then  $\text{Path}(s,t)$

Inductive rule: If  $\text{Graph}(s,u)$  and  $\text{Path}(u,t)$  then  $\text{Path}(s,t)$

## Graph

Source	Target
1	2
2	3
3	1
2	4



## Path

Source	Target
1	1
1	2
1	3
1	4
2	1
2	2
2	3
2	4
3	1
3	2
3	3
3	4

# Recursively defined temporary views in SQL

```
WITH RECURSIVE Path(source, target) AS
(
    SELECT E.source, E.target          /* Base rule */
    FROM   Graph E
    UNION
    SELECT E.source, P.target          /* Inductive rule */
    FROM   Graph E, Path P
    WHERE E.target = P.source
)
SELECT * FROM Path;
```

# Not all subqueries define SQL views!

- Consider the following SQL query:

```
SELECT E.Sid  
FROM Student S, Enroll E  
WHERE S.Sid = E.Sid AND E.Cno IN (SELECT C.Cno  
                                    FROM Course C  
                                    WHERE C.Dept = S.Major);
```

It returns the sids of all students enrolled in some course offered by the department in which he or she majors.

- Observe that the subquery

```
SELECT C.Cno  
FROM Course C  
WHERE C.Dept = S.Major
```

is **not** an SQL view, since it has a parameter **S.Major**.

# Parameterized views

- In standard SQL, subqueries such as on the previous slide can**not** be used to define **parameterized views**.
- PostgreSQL, however, permits the use **of user-defined functions that return tables**.
- Using this feature, we can specify parameterized views.

# Parameterized views: example

- In PostgreSQL, we can specify the following function returning tables:

```
CREATE FUNCTION coursesOfferedByDept(deptname TEXT)
    RETURNS TABLE(Cno TEXT) AS
$$
    SELECT C.Cno
        FROM Course C
        WHERE C.Dept = deptname;
$$ LANGUAGE SQL;
```

- The function returns a **table (relation)** of course numbers, i.e., of courses offered by the department specified by the value of the parameter **deptname**.

# Parameterized views (continued)

- The query
- ```
SELECT E.Sid  
FROM Student S, Enroll E  
WHERE S.Sid = E.Sid AND E.Cno IN (SELECT C.Cno  
FROM Course C  
WHERE C.Dept = S.Major);
```

can now be rewritten as

```
SELECT E.Sid  
FROM Student S, Enroll E  
WHERE S.Sid = E.Sid AND  
E.Cno IN (SELECT C.Cno  
FROM coursesOfferedByDept(S.Major) C);  
set of courses offered in the  
department of student S's major
```

# Parameterized views (continued)

- Incidentally, if we only want to report the courses in a specific department, we can do this by changing the subquery as follows:

```
SELECT C.Cno  
FROM coursesOfferedByDept('Math') C
```

- This query returns the courses offered by the Math department.

# SQL Functions and Expressions

# Expressions in SELECT clause

- Expressions may occur in a SELECT clause  
**SELECT expression [AS column name] (, ...)**
- The result is a relation with a single tuple

**SELECT 1 AS one;**

|     |
|-----|
| one |
| 1   |

**SELECT sqrt(2), 'John' || 'Smith', 2=3;**

| sqrt   |            |   |
|--------|------------|---|
| 1.4142 | John Smith | f |

# Expressions in SELECT clause returns a relation

- Expressions may occur in a SELECT clause  
**SELECT expression [AS column name] (, ...)**
- The result is a relation with a single tuple
- We can therefore place such an expression in a FROM clause of another query

```
SELECT q.one  
FROM  (SELECT 1 AS one) q
```

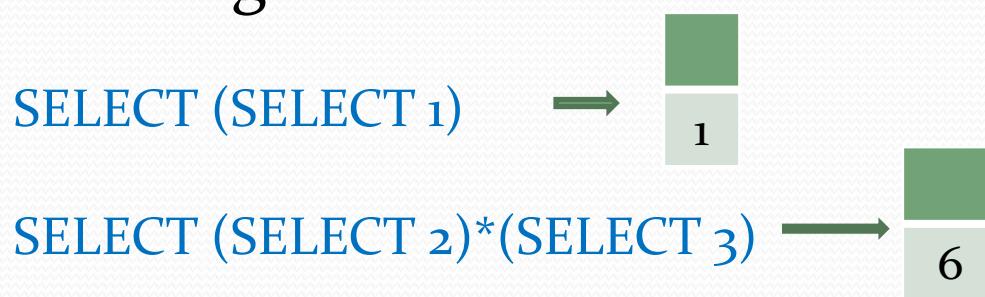
|     |
|-----|
| one |
| 1   |

# Using a **SELECT** expression inside other expressions

- To use a **SELECT expression** statement as a value in another expression, it is required to place parentheses around that expression
- The following is incorrect:

`SELECT SELECT 1`

- The following statements are correct:

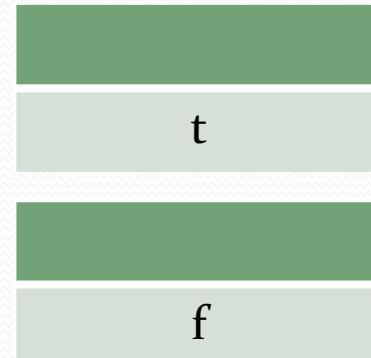


# Application: Boolean queries

- A **boolean query** is a query that returns either true (t) or false (f).
- E.g., “Are there CS students who take courses?”

```
SELECT EXISTS(SELECT S.Sid  
              FROM Student S, Enroll E  
              WHERE S.Major = ‘CS’ AND S.Sid = E.Sid);
```

- If yes, then answer is
- If no, the answer is



# Application: Checking for keys

- “Is Major a primary key for the Student relation?”
- I.e., if Major where a primary key, then there can not exist two different students with the same major.
- We can check this with the Boolean query

```
SELECT NOT EXISTS(SELECT *
                  FROM   Student S1, Student S2
                  WHERE S1.Sid <> S2.Sid AND
                        S1.Major = S2.Major) AS IsKey;
```

# Application: Checking for Foreign keys

- “Is **Sid** a foreign key in the **Enroll** relation referencing the primary key **Sid** in the **Student** relation?”
  - I.e., is it true that there does not exist a Sid value in the Enroll relation that does not appear as a Sid value in the Student relation?

## Employee

| Eid | Salary |
|-----|--------|
| p1  | 10000  |
| p2  | 10000  |
| p3  | 20000  |

- “Find new employee salaries after an increase by 5%.”

```
SELECT E.Eid, E.Salary*1.05 AS NewSalary  
FROM Employee E
```

| Eid | NewSalary |
|-----|-----------|
| p1  | 10500     |
| p2  | 10500     |
| p3  | 21000     |

| Eid | Salary |
|-----|--------|
| p1  | 10000  |
| p2  | 10000  |
| p3  | 20000  |

- “Report whether or not an employee has the lowest salary.”

```

SELECT E.Eid, E.Salary ≤ ALL(SELECT E1.Salary
                             FROM Employee E1) AS
      HasLowestSalary
FROM Employee E
    
```

| Eid | HasLowestSalary |
|-----|-----------------|
| p1  | t               |
| p2  | t               |
| p3  | f               |

# CASE expression

```
SELECT E.Eid,  
       CASE WHEN E.Salary > 100000 THEN 'high'  
             WHEN E.Salary < 10000   THEN 'low'  
             ELSE 'medium'  
         END AS SalaryRange  
FROM Employee E;
```

| Eid | Salary |
|-----|--------|
| p1  | 25000  |
| p2  | 9000   |
| p3  | 105000 |



| Eid | SalaryRange |
|-----|-------------|
| p1  | medium      |
| p2  | low         |
| p3  | high        |

# Expressions with parameterized sub-queries

- “Report whether or not a student takes a course in the department in which he or she majors.”

```
SELECT S.Sid, S.Major IN (SELECT C.Dept  
                           FROM Course C, Enroll E  
                           WHERE E.Sid = S.Sid AND E.Cno = C.Cno)  
FROM Student S
```

- Notice how **S.Sid** is a parameter in the subquery
- For **S.Sid**, the subquery returns the departments that offer courses in which the student with that sid is enrolled

# Expressions in WHERE clause

- Consider a relation of points in 2-dimensional space.

| Pid | X | Y |
|-----|---|---|
| p1  | 0 | 0 |
| p2  | 1 | 0 |
| p3  | 2 | 1 |
| p4  | 5 | 3 |

- “Find the pairs of points that are within distance 3.”

SELECT P1.Pid AS P1, P2.Pid AS P2

FROM Point P1, Point P2

WHERE  $\sqrt{(\text{power}(P1.X-P2.X,2)+\text{power}(P1.Y-P2.Y,2))} \leq 3$

- $\sqrt{.}$  and  $\text{power}(..)$  are PostgreSQL functions.

# Putting things together

- “Raise the salary of an employee by 5% provided that the raise is less than \$1000.”

```
(SELECT E.Eid, E.Salary*1.05 AS NewSalary  
    FROM Employee E  
   WHERE E.Salary * 0.05 < 1000)  
UNION  
(SELECT E.Eid, E.Salary AS NewSalary  
    FROM Employee E  
   WHERE E.Salary * 0.05 ≥ 1000)
```

| Eid | Salary |
|-----|--------|
| p1  | 10000  |
| p2  | 10000  |
| p3  | 20000  |

| Eid | NewSalary |
|-----|-----------|
| p1  | 10500     |
| p2  | 10500     |
| p3  | 20000     |

# Application: polynomials

- Consider the polynomial  $3x^3 - 5x + 7$
- We can represent such a polynomial in a relation  
Polynomial(coefficient int, degree int).

Polynomial

| coefficient | degree |
|-------------|--------|
| 3           | 3      |
| -5          | 1      |
| 7           | 0      |

# Derivative of polynomial

- If  $ax^n$  is a term of a polynomial and  $n \geq 1$  then its derivative is  $n * ax^{n-1}$
- The derivative of the previous polynomial is  $3 * 3x^2 + 1 * (-5) = 9x^2 - 5$
- We can compute this derivative with an SQL query:

```
SELECT t.degree*t.coefficient AS coefficient, t.degree-1 AS degree  
FROM Polynomial t  
WHERE t.degree >= 1
```

| coefficient | degree |
|-------------|--------|
| 9           | 2      |
| -5          | 0      |

# User-defined functions

- Users can define their own functions and then use them in expressions.
- “Define the increment-by 1 function.”

```
CREATE FUNCTION increment (n INTEGER)
RETURNS INTEGER AS
$$
SELECT n+1;
$$ LANGUAGE SQL;
```

- This function can now be used in a query. E.g.,

```
SELECT increment(5) AS Value;
```

| Value |
|-------|
| 6     |

# User-defined functions

- Write the **distance** function between two points

$$(x_1, y_1) \text{ and } (x_2, y_2): \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

```
CREATE FUNCTION distance(x1 FLOAT, y1 FLOAT, x2 FLOAT, y2 FLOAT)
RETURNS FLOAT AS
$$
SELECT sqrt(power(x1-x2,2)+power(y1-y2,2));
$$ LANGUAGE SQL;
```

| Pid | X | Y |
|-----|---|---|
| p1  | 0 | 0 |
| p2  | 1 | 0 |
| p3  | 2 | 1 |
| p4  | 5 | 3 |

- “Find the pairs of points that are within distance 3.”

```
SELECT P1.Pid AS P1, P2.Pid AS P2
FROM Point P1, Point P2
WHERE distance(P1.X,P1.Y,P2.X,P2.Y) ≤ 3
```

# Functions with output parameters

- Functions can take input parameters
- Functions can also have **output parameters** (OUT)
- OUT parameters are useful to return tuples and relations with several attributes named by these parameters.

```
CREATE FUNCTION sum_and_product(x int, y int, OUT sum int, OUT product int)
AS
$$
  SELECT x+y, x*y;
$$ LANGUAGE SQL;
```

- Example,

```
SELECT t.sum, t.product
FROM sum_and_product(3,4) t;
```

| sum | product |
|-----|---------|
| 7   | 12      |

# Functions with output parameters return tuples

```
CREATE FUNCTION sum_and_product(x int, y int, OUT sum int, OUT product int)
AS
$$
  SELECT x+y, x*y;
$$ LANGUAGE SQL;
```

SELECT sum\_and\_product(3,4); →

| sum_and_product |
|-----------------|
| (7,12)          |

SELECT (sum\_and\_product(3,4)).sum, (sum\_and\_product(3,4)).product



| sum | product |
|-----|---------|
| 7   | 12      |

# Record (tuple) types

- SQL permits the definition of record (tuple) types.
- Functions can return a record.

```
CREATE TYPE edge AS (source INT, target INT);
```

```
CREATE FUNCTION printEdge(x INT, y INT) RETURNS edge  
AS $$  
    SELECT x, y;  
$$ LANGUAGE SQL;
```

```
SELECT * FROM printEdge(1,2);
```



| source | target |
|--------|--------|
| 1      | 2      |

```
SELECT source FROM printEdge(1,2);
```



| source |
|--------|
| 1      |

# Functions returning sets (relations)

- A function can also return a set of tuples (relation).
- The return type of such a function is specified using the **RETURN SET record** clause.
- Let **Pair(x int, y int)** be a relation of pairs:

```
CREATE FUNCTION sum_and_product(OUT sum int, OUT product int)
RETURNS SETOF RECORD
AS $$  
    SELECT P.x+P.y, P.x*P.y FROM Pair P;  
$$ LANGUAGE SQL;
```

- Can be used for user-defined views.

# Alternative RETURNS TABLE

```
CREATE FUNCTION sum_and_product()
RETURNS TABLE (sum INTEGER, product INTEGER)
AS $$  
    SELECT P.x+P.y, P.x*P.y FROM Pair P;  
$$ LANGUAGE SQL;
```

Pair

| x | y  |
|---|----|
| 1 | 2  |
| 3 | -4 |

```
SELECT t.sum, t.product  
FROM sum_and_product() t;
```

| sum | product |
|-----|---------|
| 3   | 2       |
| -1  | -12     |

# Caution: non determinism

- Functions returning a record return a single record even if the body of the function computes a set of record.
- Leads to **non-deterministic** effects.

```
CREATE FUNCTION sum_and_product(OUT sum int, OUT product int)
RETURNS RECORD
AS $$  
    SELECT P.x+P.y, P.x*P.y FROM Pair P ORDER BY RANDOM();
$$ LANGUAGE SQL;
```

```
SELECT * FROM sum_and_product();
```

| sum | product |
|-----|---------|
| 3   | 2       |

or

| sum | product |
|-----|---------|
| -1  | -12     |

# Aggregate Functions and Data Partitioning

# Collections and aggregate functions

- A **collection** is a grouping of some variable number of data items (possibly zero)
- Usually the data items in a collection are of the **same type**
- **Aggregate functions** are functions that apply to collections, i.e., they consider **all** these data items in these collections
- Applied to a collection, an aggregate function returns a single value

# Examples of collections

- sets, multisets, dictionaries (maps), relations
- vectors, lists, arrays, series
- data structures: stacks, queues, hash tables, trees, graphs

# Aggregate functions on unordered collections

- We will restrict ourselves to aggregate functions on sets, bags, and relations
- COUNT (we will often use the notation  $|A|$  instead of  $\text{COUNT}(A)$ )
- SUM, AVERAGE, MIN, and MAX
- $\text{COUNT}(\{a,b,c\}) = 3$ ;  $\text{COUNT}(\{\{a,c,c,b,b,b\}\}) = 6$
- $\text{SUM}(\{1, 4, 7\}) = 12$ ;  $\text{SUM}(\{\{1,1,1,1,1\}\}) = 5$
- $\text{AVG}(\{1,4,7\}) = 4$ ;  $\text{AVG}(\{\{1,1,1,1,1\}\}) = 1$
- $\text{MIN}(\{1,4,7\}) = \text{MIN}(\{\{1,1,1,1,1\}\}) = 1$
- $\text{MAX}(\{1,4,7\}) = 7$ ;  $\text{MAX}(\{\{1,1,1,1,1\}\}) = 1$
- $\text{MIN}(\{\text{John}, \text{Eric}, \text{Ann}\}) = \text{Ann}$
- These aggregate functions are supported in SQL

# Applications of aggregate functions

- Data analytics
- Formulating complex queries
- Facilitating efficient query evaluation

# Counting the size of a set in SQL

- Let  $R(A_1, \dots, A_n)$  be a relation.
- Then  $|R|$  can be obtained as follows:

```
SELECT COUNT(*)  
FROM   R r;
```

- Alternatively,

```
SELECT COUNT(1)  
FROM   R r;
```

- Of course we can restrict the **COUNT** function to apply to a subset of  $R$  by applying a **WHERE** clause.

# COUNT examples

- “Find the number of courses in which the student with sid 10 is enrolled.”

```
SELECT COUNT(*)  
FROM Enroll E  
WHERE E.sid = ‘s10’;
```

- “Find the number of students who are not enrolled in any CS course.”

```
SELECT COUNT(*)  
FROM Student S  
WHERE S.Sid NOT IN (SELECT E.Sid  
FROM Enroll E, Course C  
WHERE E.Cno = C.Cno AND C.Dept = ‘CS’);
```

# COUNT Example

- Let R and S be two relations, then the following query will return  $|R \times S| = |R||S|$ , i.e. the size of the cartesian (cross) product of R and S.

```
SELECT COUNT(*)  
FROM R r, S s;
```

- $R \times S = \{(r, s) \mid r \in R \wedge s \in S\}$

# COUNT DISTINCT

R

| A | B |
|---|---|
| a | 1 |
| a | 2 |
| b | 1 |

SELECT COUNT(r1.A) AS Total  
FROM R r1, R r2



| Total |
|-------|
| 9     |

SELECT COUNT(DISTINCT r1.A) AS Total  
FROM R r1, R r2



| Total |
|-------|
| 2     |

# Simulating COUNT with SUM

- The following SQL query uses the SUM aggregate function to determine the size of R, provided  $R \neq \emptyset$

```
SELECT SUM(1)
  FROM   R r;
```

- The bag that is generated by the query is  $\{\{1,\dots,1\}\}$  containing as many 1's as there are tuples  $r$  in  $R$

# Example

- Consider the relation R
- $|R| = 3$
- COUNT applies to  $\{\{a,b,c\}\}$
- SUM applies to  $\{\{1, 1, 1\}\}$
- If you write the SQL query

```
SELECT SUM(2)
FROM   R r;
```

| A |
|---|
| a |
| b |
| c |

the result will be  $SUM(\{\{2,2,2\}\}) = 6$ .

# Caveat: empty collection

- Consider the relation R
- $|R| = 0$
- COUNT applied to {} gives 0
- SUM applied to {} gives NULL

```
SELECT COUNT(1)  
FROM R r;
```



|   |
|---|
| A |
|---|

```
SELECT SUM(1)  
FROM R r;
```



|   |
|---|
| 0 |
|---|

|      |
|------|
| NULL |
|------|

# MIN and MAX aggregate functions

- **MIN** returns the smallest data item in the bag to which it applies.
- **MAX** returns the largest data item in the bag to which it applies.
- Data items can come from any ordered basic domain: integer, float, text
- A more general **MIN** function can be simulated using  $\leq \text{ALL}$

```
SELECT DISTINCT r.A1,...,r.An  
FROM R r  
WHERE (r.A1,...,r.An)  $\leq \text{ALL}$  (SELECT r1.A1,...,r1.An  
FROM R r1);
```

- **MAX** can be simulated using  $\geq \text{ALL}$
- However, there is a problem if MIN (MAX) is applied to an empty set.

# CAVEAT: aggregate functions on empty set

- Except for COUNT, SQL aggregate functions return a NULL value when applied to an empty set (or bag).
- Assume R is the empty relation A
- Then `SELECT MIN(r.A) AS smallest FROM R r`  
returns the relation smallest  
NULL
- However `SELECT r.A AS smallest FROM R r WHERE r.A <= ALL(SELECT r1.A FROM R r1)`  
returns the empty relation smallest

# Partitioning and counting

- “Determine for each student the number of courses taken by that student.”

Enroll

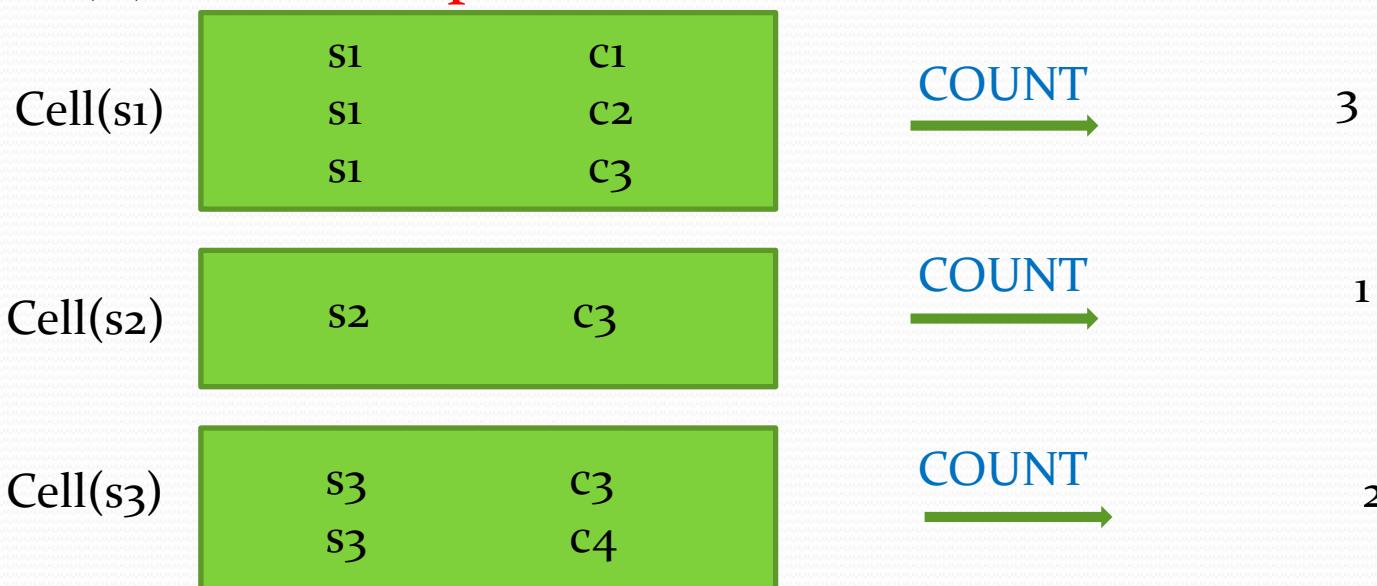
| Sid | Cno |
|-----|-----|
| s1  | c1  |
| s2  | c1  |
| s1  | c2  |
| s3  | c3  |
| s3  | c4  |
| s1  | c3  |



| Sid | No_Courses |
|-----|------------|
| s1  | 3          |
| s2  | 1          |
| s3  | 2          |

# Partition and map count function

- (1) First, **Partition** the Enroll table into cells (blocks) wherein each cell contains all the tuples that have a common sid value.
- (2) Next, **Map** the **COUNT** function over these cells.



# Partition and map COUNT in SQL

- (1) The **GROUP BY** map **COUNT** method
- (2) The user-defined **COUNT FUNCTION** method
- (3) The **SELECT COUNT-expression** method

# The GROUP BY map COUNT method

Map COUNT phase → `SELECT E.Sid, COUNT(*) AS No_Courses  
FROM Enroll E  
GROUP BY(E.Sid)`

**Partition phase:** the GROUP BY operator places each tuple E into the cell identified by its E.Sid value

**Map COUNT phase:** the COUNT function is mapped over the cells identified by the different possible E.sid values

# Group by visualization

```
SELECT State, COUNT(*)  
FROM facebook  
GROUP BY State
```

facebook

| Name  | State | # of Friends |
|-------|-------|--------------|
| Matt  | CA    | 300          |
| Lisa  | CA    | 500          |
| Jeff  | CA    | 600          |
| Sarah | FL    | 400          |

final table

| State | COUNT(*) |
|-------|----------|
| CA    | 0        |
| FL    | 0        |

# The user-defined COUNT FUNCTION method

- “Define a function with input parameter a student sid and as output the number of courses taken by that student.

```
CREATE FUNCTION NumberOfCourses (s TEXT) RETURNS bigint  
AS $$
```

```
    SELECT COUNT(*)  
        FROM Enroll E  
       WHERE E.Sid = s;  
$$ LANGUAGE SQL;
```



Map COUNT phase

- Then execute the SQL query

```
SELECT S.Sid, NumberOfCourses(S.Sid) AS No_Courses  
FROM Student S;
```



Partition phase

## Student

| Sid |
|-----|
| s1  |
| s2  |
| s3  |
| s4  |

## Enroll

| Sid | Cno |
|-----|-----|
| s1  | c1  |
| s1  | c2  |
| s1  | c3  |
| s2  | c3  |
| s3  | c3  |
| s3  | c4  |

```
SELECT S.Sid, NumberOfCourses(S.Sid) AS No_Courses  
FROM Student S;
```



| Sid | No_Courses |
|-----|------------|
| s1  | 3          |
| s2  | 1          |
| s3  | 2          |
| s4  | 0          |

student s4 takes no courses



# The SELECT COUNT-expression method

```
SELECT S.sid, (SELECT COUNT(E.Cno) AS NumberCourses  
    FROM Enroll E  
   WHERE E.Sid = S.Sid)  
FROM Student S
```

Map phase

Partition phase

- Observe that the subquery identified by S.Sid appears in the outer SELECT clause.
- The COUNT of the result of this subquery is then delivered as a value in the outer SELECT clause.
- Notice that this expression must appear between parentheses.
- The output of this query is the same as that on the previous slide.

# Example query

- “Find the sid of each student who takes the most courses.”
- Using the COUNT FUNCTION method:

```
SELECT S.sid  
FROM Student S  
WHERE NumberOfCourses(S.sid) >= ALL (SELECT NumberOfCourses(S1.Sid)  
                                FROM Student S1);
```

# Example query

- “Find the sid of each student who take the most courses.”
- Using the GROUP BY method:

WITH

```
NumberOfCoursesbyStudent AS (SELECT E.Sid, COUNT(E.Cno) As NumberOfCourses  
    FROM Enroll E  
    GROUP BY(E.Sid))
```

```
SELECT P.sid  
FROM NumberofCoursesbyStudent P  
WHERE P.NumberofCourses >= ALL (SELECT P1.NumberofCourses  
    FROM NumberofCoursesbyStudent P1);
```



# The COUNT-bug of GROUP BY

- The result of the following 2 queries is the same. Notice that there is a bug since, if a student sid takes no courses, then (sid, 0) does not appear in the output.

```
SELECT E.Sid, COUNT(E.Cno)
FROM Enroll E
GROUP BY (E.Sid)
```

```
SELECT S.Sid, Count(E.Cno)
FROM Student S, Enroll E
WHERE S.Sid = E.Sid
GROUP BY(S.Sid)
```

- These two queries give the same result and exhibit the COUNT bug: the tuple (s4,0) does not appear in the result.

# Fixing the COUNT-bug

- To fix the COUNT-bug we need to add the  $(s, 0)$  pair if student with sid  $s$  takes no courses. This can be done with the **UNION** operator.

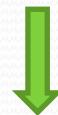
```
(SELECT E.Sid, COUNT(E.Cno) AS No_Courses  
FROM Enroll E  
GROUP BY (E.Sid))  
UNION  
(SELECT S.Sid, 0 AS No_Courses  
FROM Student S  
WHERE S.Sid NOT IN (SELECT E.Sid  
FROM Enroll E))
```

# Partitioning on different dimensions

Enroll

| Sid | Cno |
|-----|-----|
| s1  | c1  |
| s1  | c2  |
| s1  | c3  |
| s2  | c3  |
| s3  | c3  |
| s3  | c4  |

SELECT COUNT(\*)  
FROM Enroll  
GROUP BY ()



6

SELECT E.Sid, E.Cno, COUNT(\*)  
FROM Enroll E  
GROUP BY (E.Sid, E.Cno)



SELECT E.Sid, COUNT(\*)  
FROM Enroll E  
GROUP BY (E.Sid)



| E.sid |   |
|-------|---|
| s1    | 3 |
| s2    | 1 |
| s3    | 2 |

| E.Sid | E.Cno |   |
|-------|-------|---|
| s1    | c1    | 1 |
| s1    | c2    | 1 |
| s1    | c3    | 1 |
| s2    | c3    | 1 |
| s3    | c3    | 1 |
| s3    | c4    | 1 |

# What can appear in the GROUP BY clause?

- Answer: any valid expression over the tuples in the FROM clause.

```
SELECT ...  
FROM R1 t1,...,Rn tn  
WHERE ...  
GROUP BY(expression(t1,...,tn))
```

- Partition: there will be as many cells in the partition as there are different values for **expression(t<sub>1</sub>,...,t<sub>n</sub>)**

# Example: expressions in GROUP BY

| S |   |
|---|---|
| X | Y |
| 2 | 3 |
| 1 | 3 |
| 2 | 1 |
| 0 | 3 |

```
SELECT s.x + s.y AS sum, COUNT(*) as cell_size  
FROM S s  
GROUP BY (s.x + s.y)
```



| sum | cell_size |
|-----|-----------|
| 3   | 2         |
| 4   | 1         |
| 5   | 1         |

Notice that there are 4 tuples in R assigned to variable s but only 3 different s.x+s.y values: 3, 4, 5. Thus there are only 3 cells in the partition.

# Example: expression in GROUP BY

| Person |     |
|--------|-----|
| Pid    | Age |
| p1     | 10  |
| p2     | 9   |
| p3     | 12  |
| p3     | 9   |

```
SELECT p.age > 10 AS OlderThanTen, COUNT(*)  
FROM Person p  
GROUP BY (p.age > 10);
```



| OlderThanTen | Count |
|--------------|-------|
| f            | 3     |
| t            | 1     |

# Restrictions on SELECT clause in GROUP BY query

- In a GROUP BY query, the SELECT clause may only contain **aggregate expressions** that returns a **single value** for each cell of the partition induced by the GROUP BY clause.

```
SELECT AggregateExpression(cell(expression(t1,...,tn)), ...  
FROM R t1,...,tn  
WHERE condition(t1,...,tn)  
GROUP BY (expression(t1,...,tn))
```

# Aggregate expressions in SELECT clause

| S |   |
|---|---|
| X | Y |
| 2 | 3 |
| 1 | 3 |
| 2 | 1 |
| 0 | 3 |

```
SELECT s.x+s.y AS sum,  
       SUM(s.x*s.y) AS  
             sum_of_products  
FROM   S s  
GROUP BY (s.x+s.y)
```



| sum | sum_of_products       |
|-----|-----------------------|
| 3   | $2 = (2 * 1 + 0 * 3)$ |
| 4   | $3 = (1 * 3)$         |
| 5   | $6 = (2 * 3)$         |

# Aggregate expressions in SELECT clause

- The following query will raise an error since s.x is **not necessarily unique** in a cell defined by s.x+s.y values

S

| X | Y |
|---|---|
| 2 | 3 |
| 1 | 3 |
| 2 | 1 |
| 0 | 3 |

```
SELECT s.x  
FROM S s  
GROUP BY (s.x+s.y)
```

# The HAVING clause in GROUP BY queries

- The HAVING clause in a GROUP BY selects those cells from the partition induced by the GROUP BY clause that satisfy an **Aggregate Condition**.
- Only those cells are passed onto the SELECT clause.

```
SELECT AggregateExpression(Cell(expression(t1,...,tn)), ...  
FROM R1 t1, ..., R tn  
WHERE condition(t1,...,tn)  
GROUP BY (expression(t1,...,tn))  
HAVING AggregateCondition(Cell(expression(t1,...,tn)))
```

# Example: HAVING clause

- “For each student who majors in CS determine the number of courses taken by that student, provided that this number is at least 2.”

```
SELECT E.Sid, COUNT(E.Cno)
FROM Enroll E, Student S
WHERE E.Sid = S.Sid AND S.Major = ‘CS’
GROUP BY (E.Sid)
HAVING COUNT(E.Cno) ≥ 2;
```

# Simulating HAVING clause with user-defined functions in WHERE

- “For each student who majors in CS, determine the number of courses taken by that student, provided that this number is at least 3.”
- The HAVING condition can be simulated in the WHERE clause with user-defined functions.

```
SELECT S.Sid AS Sid, NumberOfCourses(S.Sid)
FROM   Student S
WHERE  S.major = 'CS' AND NumberOfCourses(S.Sid) ≥ 3
```

# Spreadsheet (Data Cube)

| Sid | Cno |
|-----|-----|
| s1  | c1  |
| s1  | c2  |
| s1  | c3  |
| s2  | c1  |
| s2  | c2  |
| s3  | c2  |
| s4  | c1  |

|          | c1 | c2 | c3 |  | sum(sid) |       |
|----------|----|----|----|--|----------|-------|
| s1       | 1  | 1  | 1  |  | 3        |       |
| s2       | 1  | 1  | 0  |  | 2        |       |
| s3       | 0  | 1  | 0  |  | 1        |       |
| s4       | 1  | 0  | 0  |  | 1        |       |
| sum(cno) | 3  | 3  | 1  |  |          | sum() |
|          |    |    |    |  | 7        |       |

# GROUPING sets

- It may be desirable to simultaneously generate different partitions and then apply an aggregation
- This is supported in SQL via GROUPING sets.

| Sid | Cno |
|-----|-----|
| s1  | c1  |
| s1  | c2  |
| s2  | c1  |
| s2  | c2  |
| s3  | c1  |



| Sid | Cno | Count |
|-----|-----|-------|
| s1  | *   | 2     |
| s2  | *   | 2     |
| s3  | *   | 1     |
| *   | c1  | 3     |
| *   | c2  | 2     |

```
SELECT Sid, Cno, COUNT(*)
FROM Enroll
GROUP BY
GROUPING SETS((Sid),(Cno))
```

# **QUERIES WITH QUANTIFIERS (PART 1)**

Venn Diagram and SQL Templates



# EXAMPLE DATABASE SCHEMA

Student

| <u>sid</u> | sname | major | byear |
|------------|-------|-------|-------|
|            |       |       |       |

Course

| <u>cno</u> | cname | dept |
|------------|-------|------|
|            |       |      |

Enroll

| <u>sid</u> | <u>cno</u> | grade |
|------------|------------|-------|
|            |            |       |

Department

| <u>dept</u> | location |
|-------------|----------|
|             |          |



# QUERIES WITH QUANTIFIERS

- Find the sid of each student who takes **some** CS courses
- Find the sid of each student who takes **no** CS courses
- Find the sid of each student who takes **not only** CS courses
- Find the sid of each student who takes **only** CS courses
- Find the sid of each student who takes **not all** CS courses
- Find the sid of each student who takes **all** CS courses



# QUERIES WITH QUANTIFIERS

- Find the cno of each course that enrolls **some** Math majors
- Find the cno of each course that enrolls **no** Math majors
- Find the cno of each course that enrolls **only** Math majors
- Find the cno of each course that enrolls **all** Math majors
- Find the cno of each course that enrolls **more than half** Math majors



# QUERIES WITH QUANTIFIERS

- Find each ( $s, d$ ) pair such that student  $s$  takes **some** courses offered by department  $d$
- Find each ( $s, d$ ) pair such that student  $s$  takes **all** courses offered by department  $d$
- Find each ( $s, d$ ) pair such that student  $s$  takes **fewer than 5** courses offered by department  $d$
- ...



# EXPRESSING QUERIES WITH QUANTIFIERS IN SQL

- We will provide a general technique using **Venn-diagrams with conditions** to express queries with quantifiers
- We will use views and parameterized views to represents the sets in these Venn diagrams
- We will use the set predicates **[NOT] EXISTS** and **[NOT] IN**, and the set operations **INTERSECT** and **EXCEPT** to translate **Venn diagrams with conditions** into SQL queries that express queries with quantifiers



# QUERIES WITH QUANTIFIERS

- Find the sid of each student who takes **some** CS courses
- Find the sid of each student who takes **no** CS courses
- Find the sid of each student who takes **not only** CS courses
- Find the sid of each student who takes **only** CS courses
- Find the sid of each student who takes **not all** CS courses
- Find the sid of each student who takes **all** CS courses



# TEMPLATE FOR QUERY WITH QUANTIFIER

- Find the sid of each student who takes “quantifier” CS courses

| quantifier   |
|--------------|
| some         |
| no           |
| not only     |
| only         |
| not all      |
| all          |
| all and only |
| at least 2   |
| ...          |

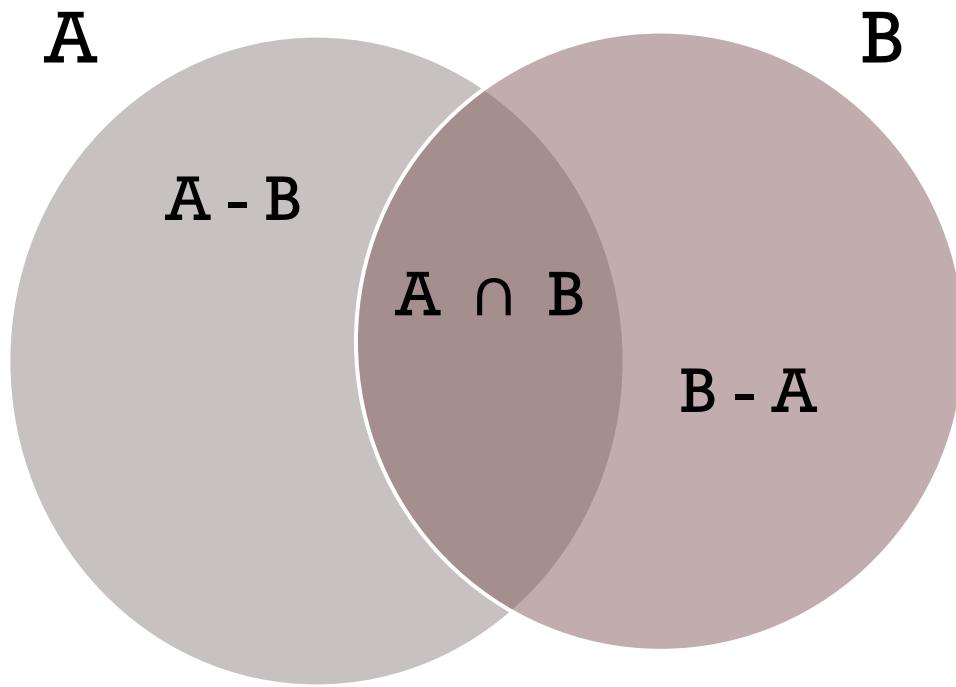


# VENN DIAGRAMS AND SQL TEMPLATES

- There is a **Venn diagram with conditions** to express a query with a quantifier
- There is a corresponding SQL statement to express this conditioned Venn diagram



# VENN DIAGRAM OF 2 SETS



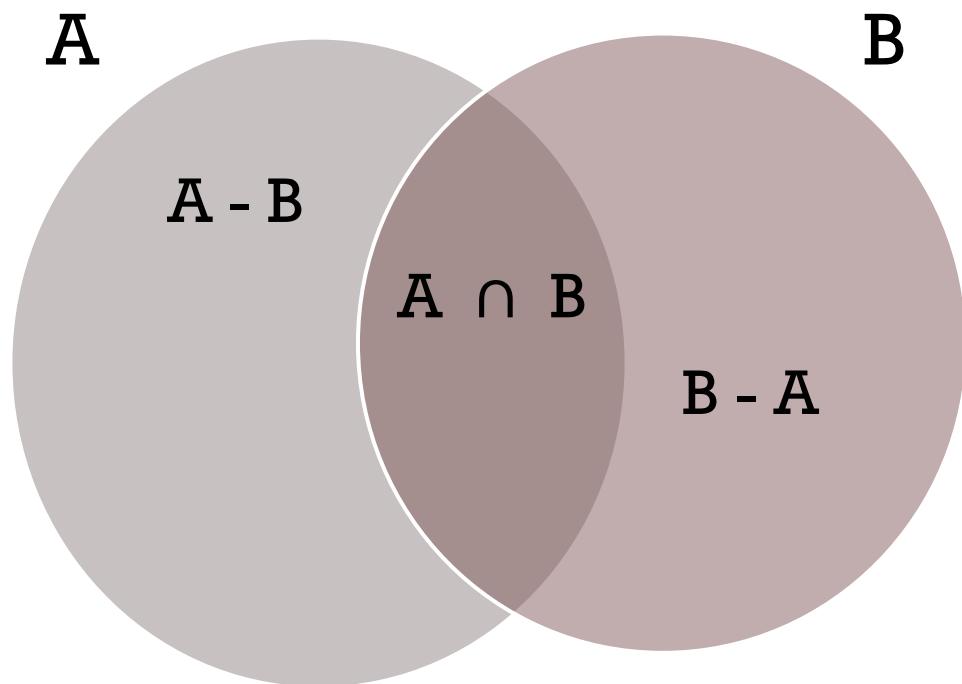
A-B Left Ear

B-A Right Ear

$A \cap B$  Lens



# VENN DIAGRAM OF 2 SETS WITH CONDITIONS



| Condition                                         |
|---------------------------------------------------|
| $A \cap B \neq \emptyset$                         |
| $A \cap B = \emptyset$                            |
| $A - B \neq \emptyset$                            |
| $A - B = \emptyset$                               |
| $B - A \neq \emptyset$                            |
| $B - A = \emptyset$                               |
| $A - B = \emptyset$<br>and<br>$B - A = \emptyset$ |
| $ A \cap B  \geq 2$                               |
| ...                                               |



# CONDITIONS EXPRESSED IN LOGIC

$$A \cap B \neq \emptyset \quad \exists x (x \in A \cap B)$$
$$\quad \quad \quad \exists x (x \in A \wedge x \in B)$$

$$A \cap B = \emptyset \quad \neg \exists x (x \in A \cap B)$$
$$\quad \quad \quad \neg \exists x (x \in A \wedge x \in B)$$



# CONDITIONS EXPRESSED IN LOGIC

$$A - B \neq \emptyset \quad \exists x (x \in A - B)$$
$$\quad \quad \quad \exists x (x \in A \wedge x \notin B)$$

$$A - B = \emptyset \quad \neg \exists x (x \in A - B)$$
$$\quad \quad \quad \neg \exists x (x \in A \wedge x \notin B)$$
$$\forall x (x \in A \rightarrow x \in B)$$
$$\quad \quad \quad A \subseteq B$$



# CONDITIONS EXPRESSED IN LOGIC

$$B - A \neq \emptyset \quad \exists x (x \in B - A)$$
$$\quad \quad \quad \exists x (x \in B \wedge x \notin A)$$

$$B - A = \emptyset \quad \neg \exists x (x \in B - A)$$
$$\quad \quad \quad \neg \exists x (x \in B \wedge x \notin A)$$
$$\forall x (x \in B \rightarrow x \in A)$$
$$\quad \quad \quad B \subseteq A$$



# CONDITIONS EXPRESSED IN LOGIC

$A - B = \emptyset$  and  $B - A = \emptyset$

$A \subseteq B$  and  $B \subseteq A$

$A = B$



# CONDITIONS EXPRESSED IN LOGIC

$$|A \cap B| \geq 2$$

$$\exists x \exists y (x \neq y \wedge x \in A \cap B \wedge y \in A \cap B)$$

$$|A \cap B| < 2$$

$$\neg \exists x \exists y (x \neq y \wedge x \in A \cap B \wedge y \in A \cap B)$$



# VENN DIAGRAM FOR OUR QUERIES

- For a student with key sid,  $\text{CoursesEnrolledIn(sid)}$  denotes the set of courses taken by that student
- $\text{CS_Courses}$  denotes the set of courses offered by the ‘CS’ department
- Linking this to a Venn diagram, we set

$A = \text{CoursesEnrolledIn}(sid)$

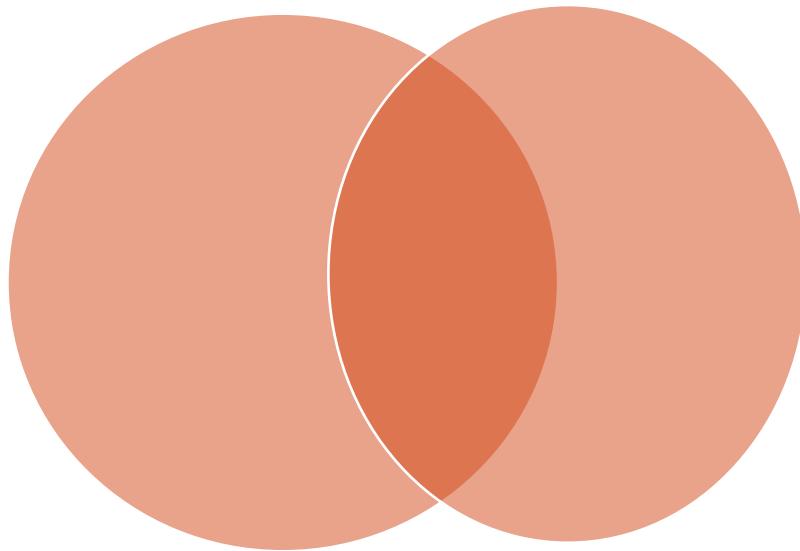
$B = \text{CS_Courses}$

- Note that for different values of sid,  $\text{CoursesEnrolledIn}(sid)$  denote different sets



# VENN DIAGRAM FOR OUR QUERIES

A = CoursesEnrolledIn(**sid**)



B = CS\_Courses

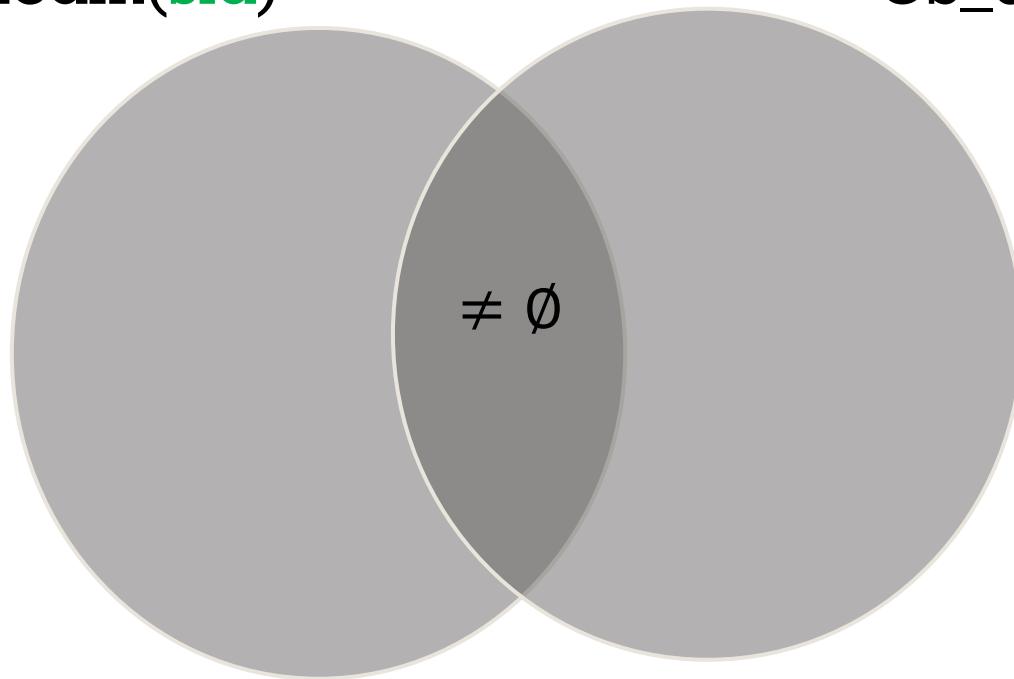


# SOME

Find the **sid** of each student who takes **some** CS courses

CoursesEnrolledIn(**sid**)

CS\_courses



# SOME

```
SELECT S.sid  
FROM Student S  
WHERE EXISTS (
```

```
    SELECT E.cno  
    FROM Enroll E  
    WHERE E.sid = S.sid  
    INTERSECT  
    SELECT C.cno  
    FROM Course C  
    WHERE C.dept = 'CS')
```

CoursesEnrolledIn(sid)

CS\_Courses



# SOME

```
SELECT S.sid
FROM Student S
WHERE EXISTS (SELECT E.cno
               FROM Enroll E
               WHERE E.sid = S.sid AND
                     E.cno IN (SELECT C.cno
                                FROM Course C
                                WHERE C.dept = 'CS'))
```



# SOME

- In SQL, the **SOME** quantifier can be expressed using the

**EXISTS ( A INTERSECT B ) template**

or using the

**EXISTS ( A IN B ) template**



# **SOME (IS A VERY SPECIAL CASE)**

```
SELECT DISTINCT E.Sid  
FROM Enroll E, Course C  
WHERE E.Cno = C.Cno AND C.Dept = 'CS'
```

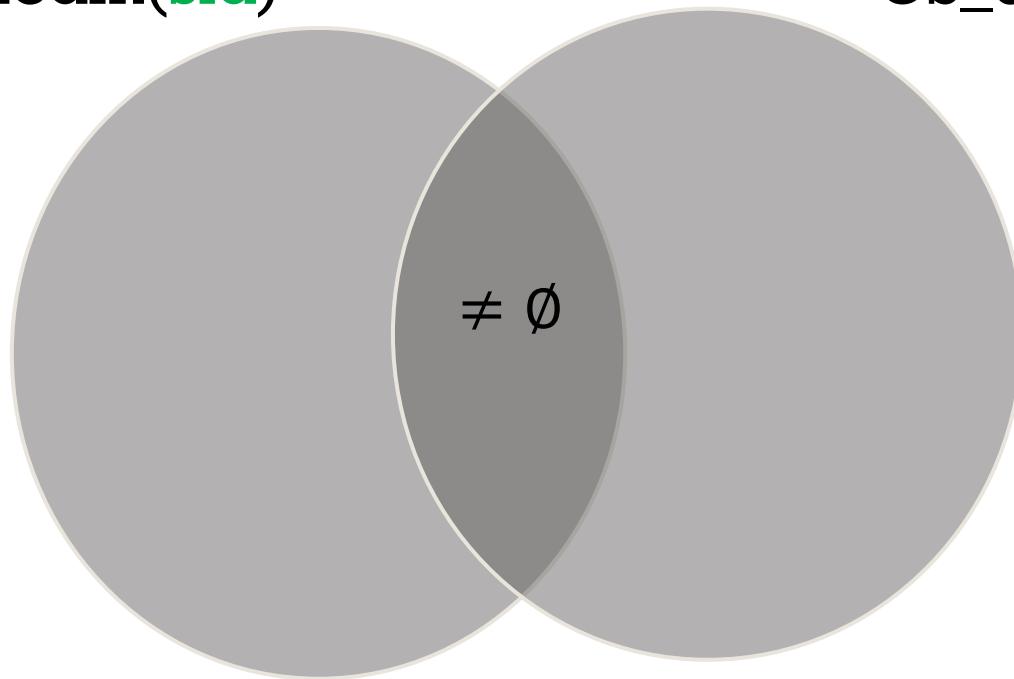


# SOME

Find the **sid** of each student who takes **some** CS courses

CoursesEnrolledIn(**sid**)

CS\_courses



# DEFINING RELEVANT SETS WITH VIEWS

- Definition of CoursesEnrolledIn(sid)

```
CREATE FUNCTION CoursesEnrolledIn(sid INTEGER)
RETURNS TABLE (cno INTEGER) AS
$$
SELECT E.cno
FROM Enroll E
WHERE E.sid = CoursesEnrolledIn.sid;
$$ LANGUAGE SQL
```

- Definition CS\_Courses

```
CREATE VIEW CS_Courses AS (SELECT C.cno
                           FROM Course C
                           WHERE C.Dept = 'CS')
```

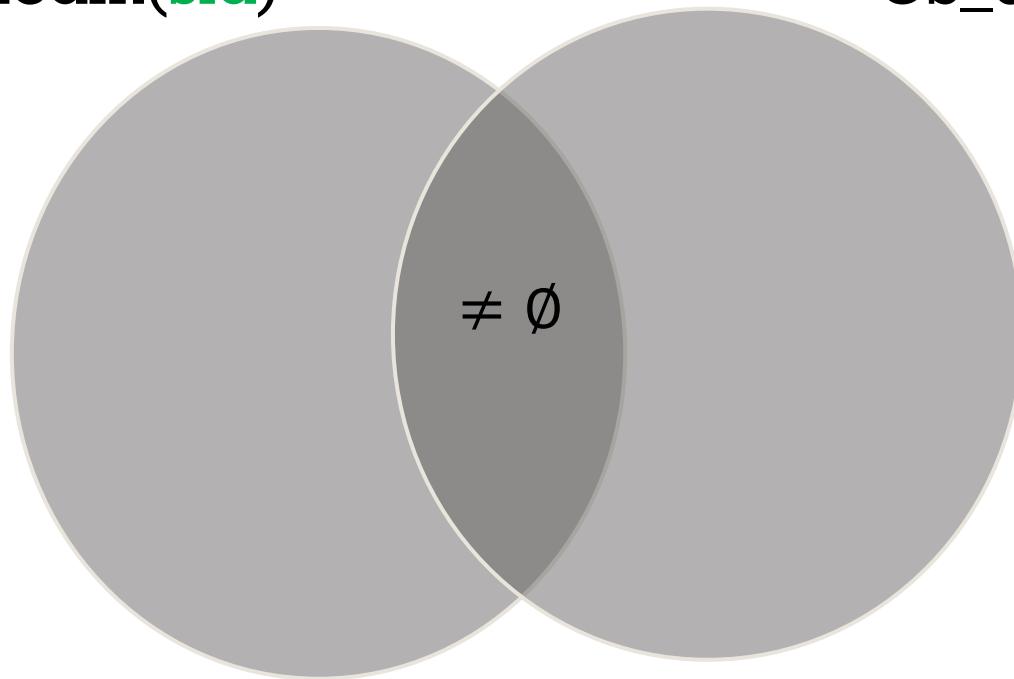


# SOME

Find the **sid** of each student who takes **some** CS courses

CoursesEnrolledIn(**sid**)

CS\_courses



# SOME

```
SELECT sid  
FROM Student  
WHERE EXISTS ( SELECT cno  
                FROM CoursesEnrolledIn(sid)  
                INTERSECT  
                SELECT cno  
                FROM CS_Courses )
```



# SOME

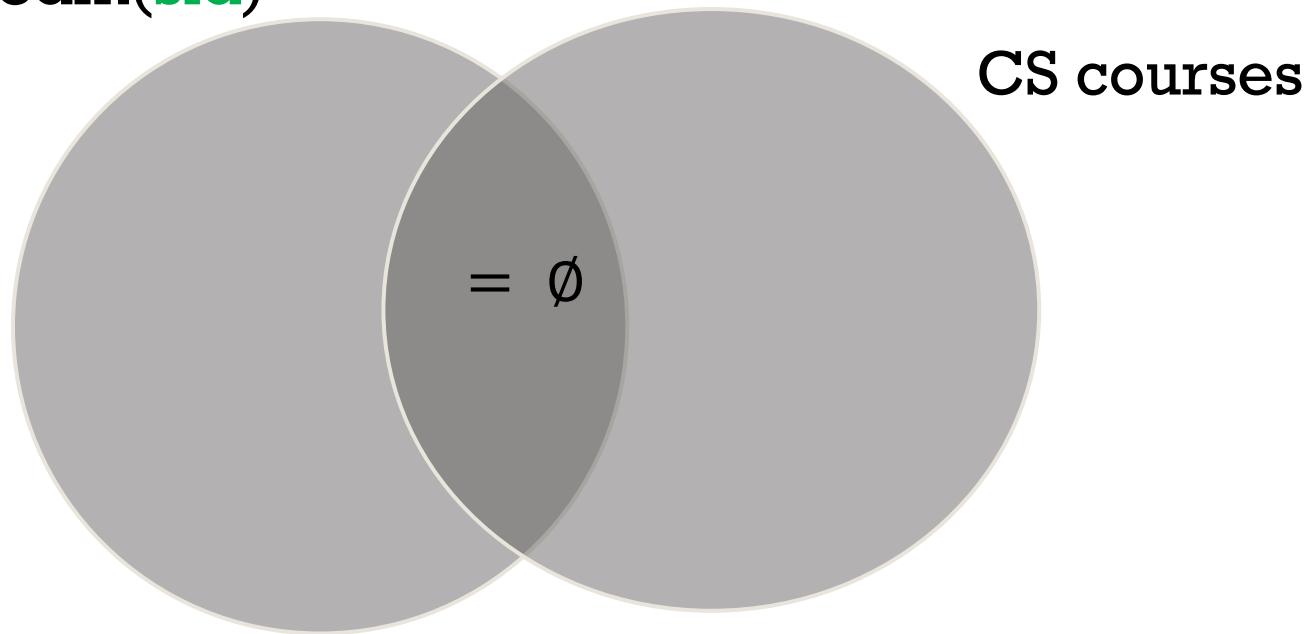
```
SELECT sid  
FROM Student  
WHERE EXISTS (SELECT cno  
                FROM CoursesEnrolledIn(sid)  
                WHERE cno IN (SELECT cno  
                               FROM CS_Courses))
```



**N**o

Find the **sid** of each student who takes **no** CS courses

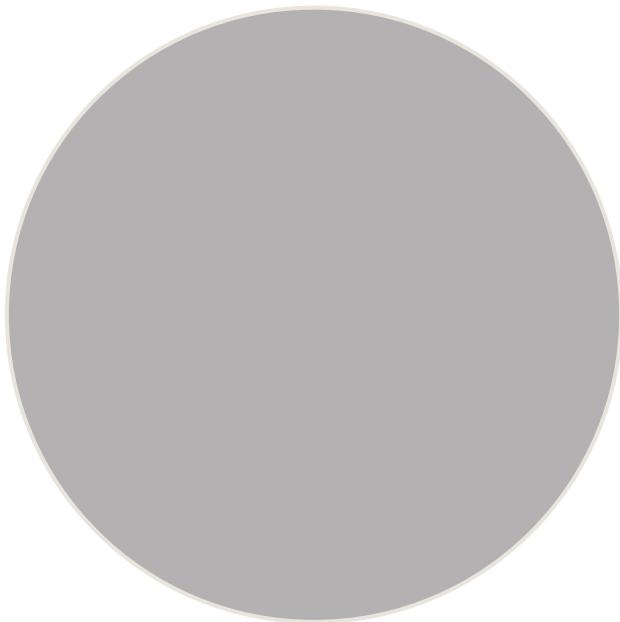
CoursesEnrolledIn(**sid**)



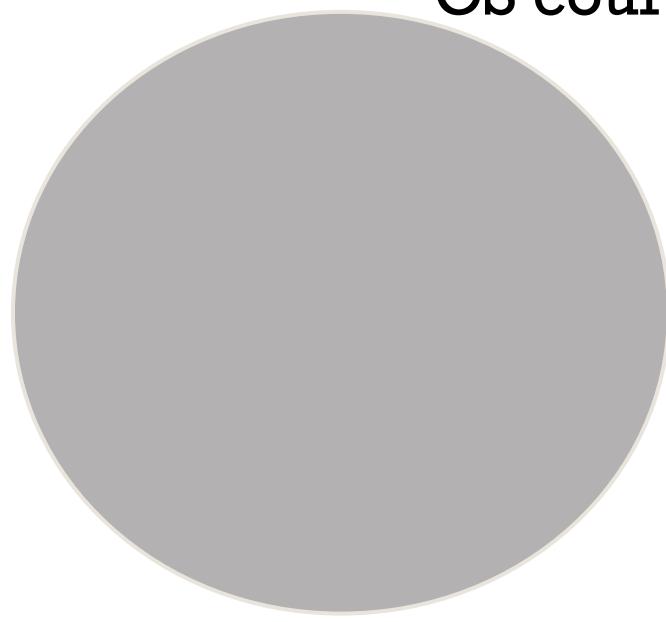
**N0**

Find the **sid** of each student who takes **no** CS courses

CoursesEnrolledIn(**sid**)



CS courses



# NO

```
SELECT sid
FROM Student
WHERE NOT EXISTS ( SELECT cno
                      FROM CoursesEnrolledIn(sid)
                      INTERSECT
                      SELECT cno
                      FROM CS_Courses )
```



# NO

```
SELECT sid  
FROM Student  
WHERE NOT EXISTS (SELECT cno  
                   FROM CoursesEnrolledIn(sid)  
                   WHERE cno IN (SELECT cno  
                                 FROM CS_Courses))
```



# NO

- In SQL, the **NO** quantifier can be expressed using the

**NOT EXISTS ( A INTERSECT B ) template**

or using the

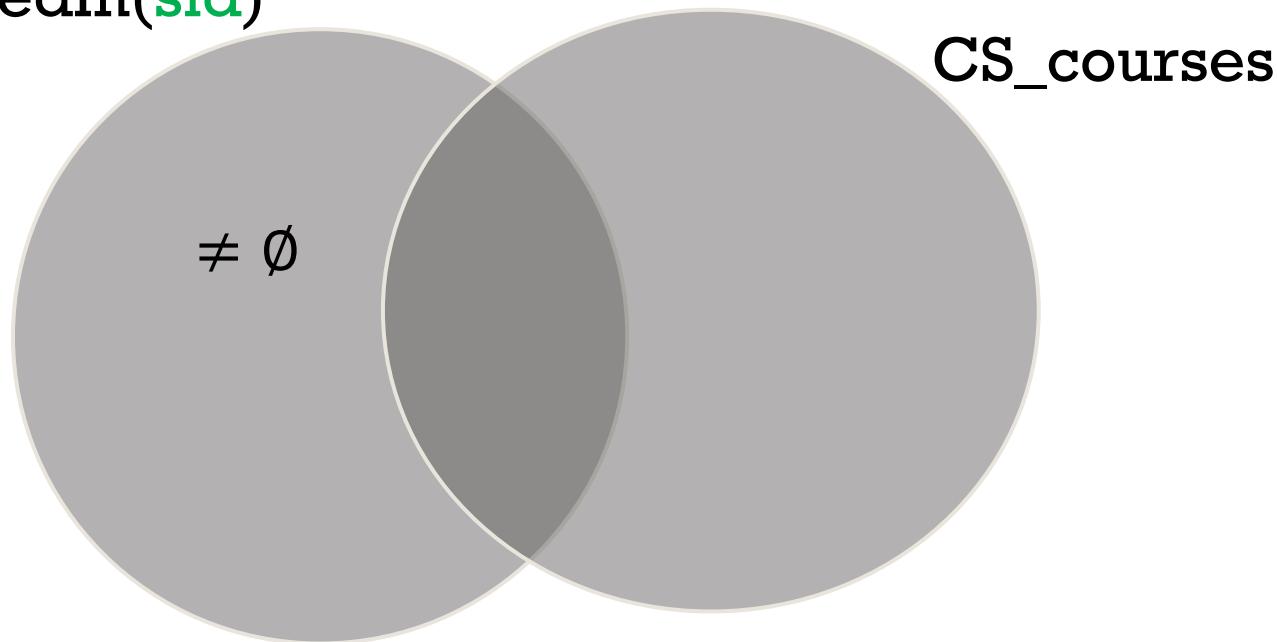
**NOT EXISTS ( A IN B ) template**



# NOT ONLY

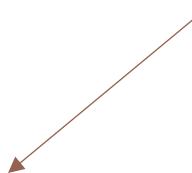
Find the **sid** of each student who takes **not only** CS cou

CoursesEnrolledIn(**sid**)



# NOT ONLY

```
SELECT sid  
FROM Student  
WHERE EXISTS ( SELECT cno  
                FROM CoursesEnrolledIn(sid)  
                EXCEPT  
                SELECT cno  
                FROM CS_Courses )
```



# NOT ONLY

```
SELECT sid  
FROM Student  
WHERE EXISTS (SELECT cno  
                FROM CoursesEnrolledIn(sid)  
                WHERE cno NOT IN (SELECT cno  
                                  FROM CS_Courses))
```



# NOT ONLY

- In SQL, the NOT ONLY quantifier can be expressed using the

EXISTS ( A EXCEPT B ) template

or using the

EXISTS ( A NOT IN B ) template



# ONLY

Find the **sid** of each student who takes **only** CS courses

CoursesEnrolledIn(**sid**)

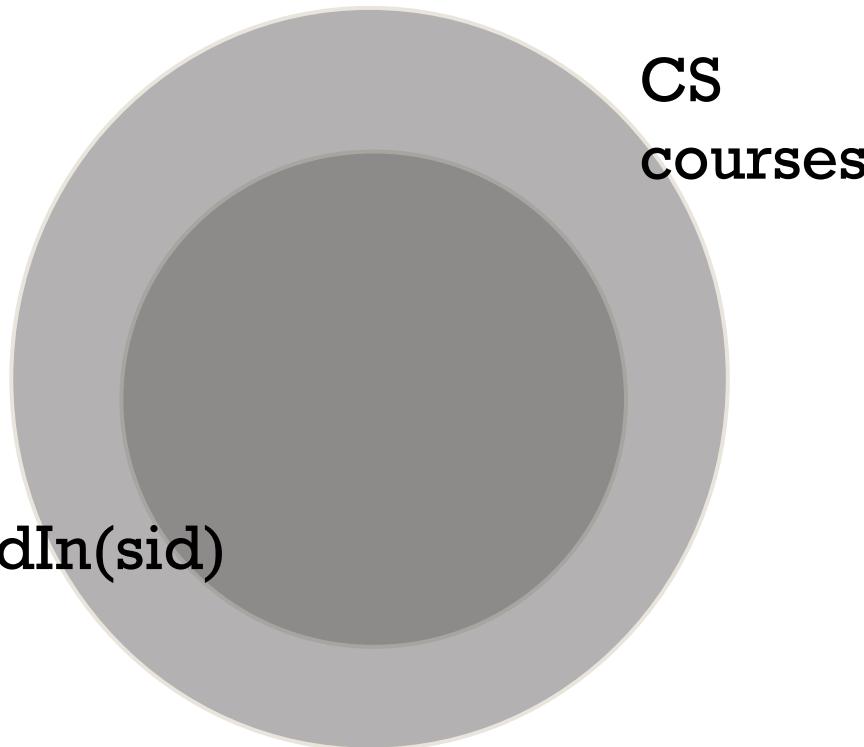
CS\_courses

$$= \emptyset$$



# ONLY

Find the **sid** of each student who takes **only** CS courses



# ONLY

```
SELECT sid  
FROM Student  
WHERE NOT EXISTS ( SELECT cno  
                    FROM CoursesEnrolledIn(sid)  
                    EXCEPT  
                    SELECT cno  
                    FROM CS_Courses )
```



# ONLY

```
SELECT sid  
FROM Student  
WHERE NOT EXISTS (SELECT cno  
                   FROM CoursesEnrolledIn(sid)  
                   WHERE cno NOT IN (SELECT cno  
                           FROM CS_Courses))
```



# ONLY

- In SQL, the **ONLY** quantifier can be expressed using the

**NOT EXISTS ( A EXCEPT B ) template**

or using the

**NOT EXISTS ( A NOT IN B ) template**



# CAUTION WITH ONLY QUANTIFIER!

- Observe that if a student with sid “**s**” takes **no** courses then  $\text{CourseEnrolledIn}(\text{s}) = \emptyset$
- In that case, the condition

```
NOT EXISTS ( SELECT cno
              FROM CoursesEnrolledIN(s)
              EXCEPT
              SELECT cno
              FROM CS_Course)
```

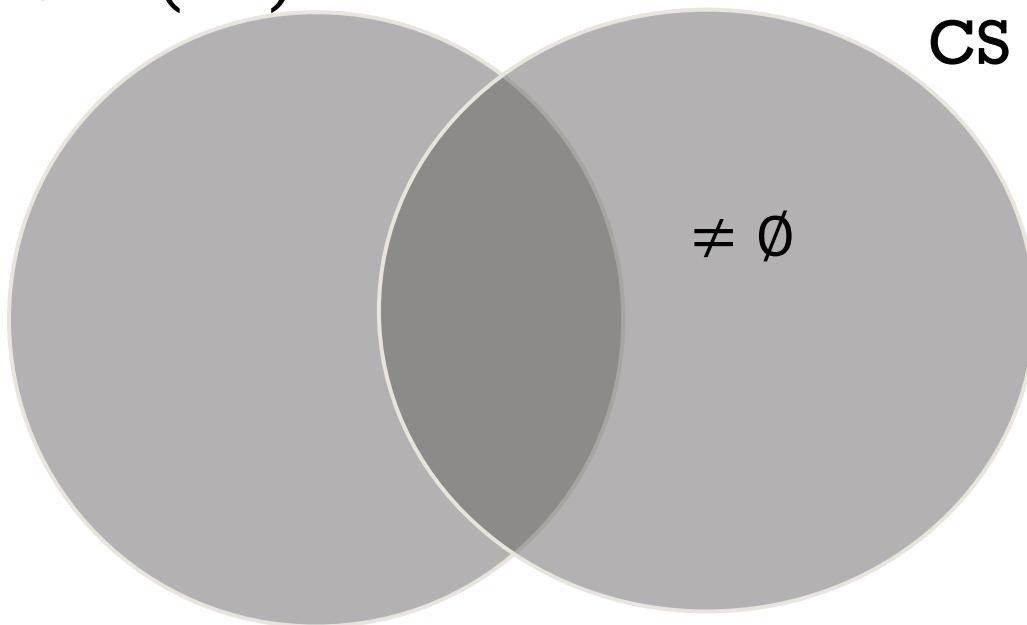
is true and therefore that student sid “**s**” is part of the solution even though that student takes no courses



# NOT ALL

Find the **sid** of each student who takes **not all** CS courses

CoursesEnrolledIn(sid)



CS courses



# NOT ALL

```
SELECT sid  
FROM Student  
WHERE EXISTS (SELECT cno  
              FROM CS_courses  
              EXCEPT  
              SELECT E.cno  
              FROM CoursesEnrolledIn(sid))
```



# NOT ALL

```
SELECT sid  
FROM Student  
WHERE EXISTS (SELECT cno  
                FROM CS_Courses  
                WHERE  
                  cno NOT IN(SELECT cno  
                            FROM CoursesEnrolledIn(sid)))
```



# NOT ALL

- In SQL, the **NOT ALL** quantifier can be expressed using the

**EXISTS ( B EXCEPT A ) template**

or using the

**EXISTS ( B NOT IN A ) template**



# ALL

Find the **sid** of each student who takes **all** CS courses

CoursesEnrolledIn(**sid**)

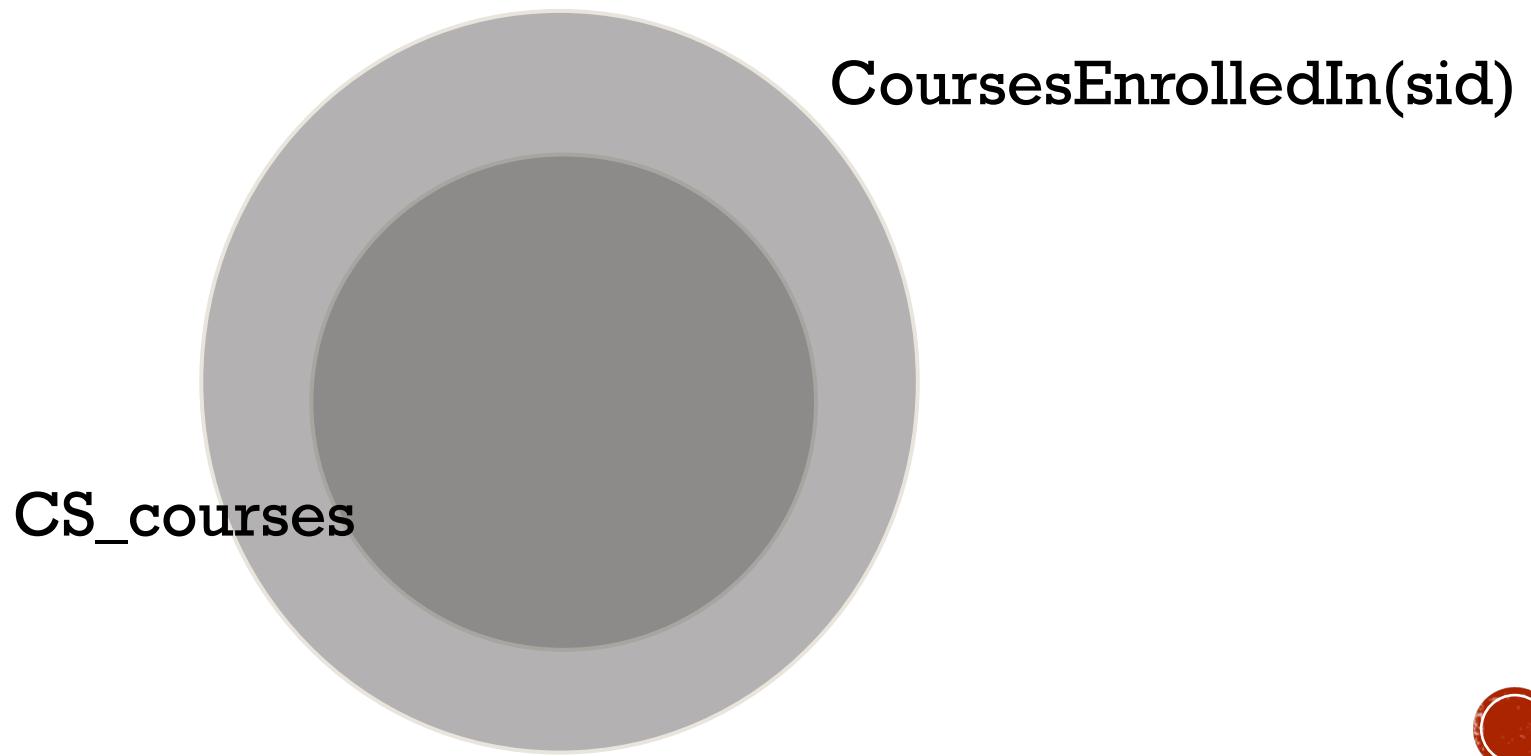
CS courses

=  $\emptyset$



# ALL

Find sid of each student who takes **all** CS courses



# ALL

```
SELECT sid
FROM Student
WHERE NOT EXISTS (SELECT cno
                   FROM CS_courses
                   EXCEPT
                   SELECT E.cno
                   FROM CoursesEnrolledIn(sid))
```



# ALL

```
SELECT sid
FROM Student
WHERE NOT EXISTS (SELECT cno
                   FROM CS_Courses
                   WHERE
                         cno NOT IN (SELECT cno
                                      FROM CoursesEnrolledIn(sid)))
```



# ALL

- In SQL, the **ALL** quantifier can be expressed using the

**NOT EXISTS ( B EXCEPT A ) template**

or using the

**NOT EXISTS ( B NOT IN A ) template**



# CAUTION WITH ALL QUANTIFIER

- Observe that if there are **no** CS courses, then a student with sid “**s**” will satisfy the **ALL** condition
- E.g, the condition

```
NOT EXISTS (SELECT cno  
            FROM  CS_Courses  
            EXCEPT  
            SELECT cno  
            FROM  CoursesEnrolledIn(s))
```

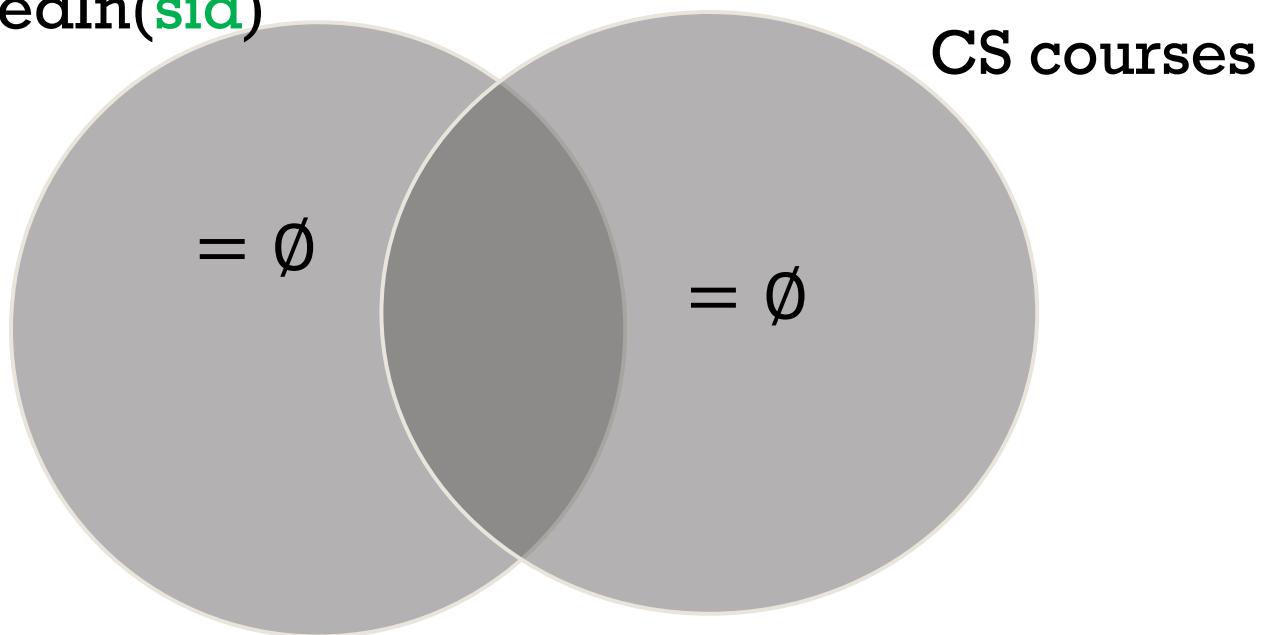
is true and therefore that student sid “**s**” is part of the solution even though there are no CS courses



# ALL AND ONLY

Find the **sid** of each student who takes **all and only** CS courses

CoursesEnrolledIn(**sid**)



# ALL AND ONLY

We have multiple quantifiers: **ALL** and **ONLY**

These must both be specified in the WHERE clause using the **ALL** and **ONLY** quantifier templates:

```
SELECT sid  
FROM Student S  
WHERE ALL template AND  
ONLY template
```



# AT LEAST TWO

Find the **sid** of each student who takes **at least two CS courses**

CoursesEnrolledIn(**sid**)

CS\_courses

$$|A \cap B| \geq 2$$



# AT LEAST 2

```
SELECT sid  
FROM Student  
WHERE EXISTS (SELECT 1  
              FROM CoursesEnrolledIn(sid) c1,  
At least two      → CoursesEnrolledIn(sid) c2  
              WHERE c1.cno <> c2.cno AND  
                    c1.cno IN (SELECT cno  
                                FROM CS_Courses) AND  
                    c2.cno IN (SELECT cno  
                                FROM CS_Courses))
```



# **QUERIES WITH QUANTIFIERS (PART 2)**

Venn Diagram and SQL Templates



# EXAMPLE DATABASE SCHEMA

Student

| <u>sid</u> | sname | major | byear |
|------------|-------|-------|-------|
|            |       |       |       |

Course

| <u>cno</u> | cname | dept |
|------------|-------|------|
|            |       |      |

Enroll

| <u>sid</u> | <u>cno</u> | grade |
|------------|------------|-------|
|            |            |       |

Department

| <u>dept</u> | location |
|-------------|----------|
|             |          |



# QUERIES WITH QUANTIFIERS

- Find the sid of each student who takes **some** CS courses
- Find the sid of each student who takes **no** CS courses
- Find the sid of each student who takes **not only** CS courses
- Find the sid of each student who takes **only** CS courses
- Find the sid of each student who takes **not all** CS courses
- Find the sid of each student who takes **all** CS courses



# QUERIES WITH QUANTIFIERS

- Find each ( $s, d$ ) pair such that student  $s$  takes **some** courses offered by department  $d$
- Find each ( $s, d$ ) pair such that student  $s$  takes **all** courses offered by department  $d$
- Find each ( $s, d$ ) pair such that student  $s$  takes **fewer than 5** courses offered by department  $d$
- ...



# QUERIES WITH QUANTIFIERS RETURNING PAIRS

Find each ( $s, d$ ) pair such that student  $s$  takes ‘quantifier’ courses offered by department  $d$ .

| quantifier   |
|--------------|
| some         |
| no           |
| not only     |
| only         |
| not all      |
| all          |
| all but only |
| at least 2   |
| ...          |

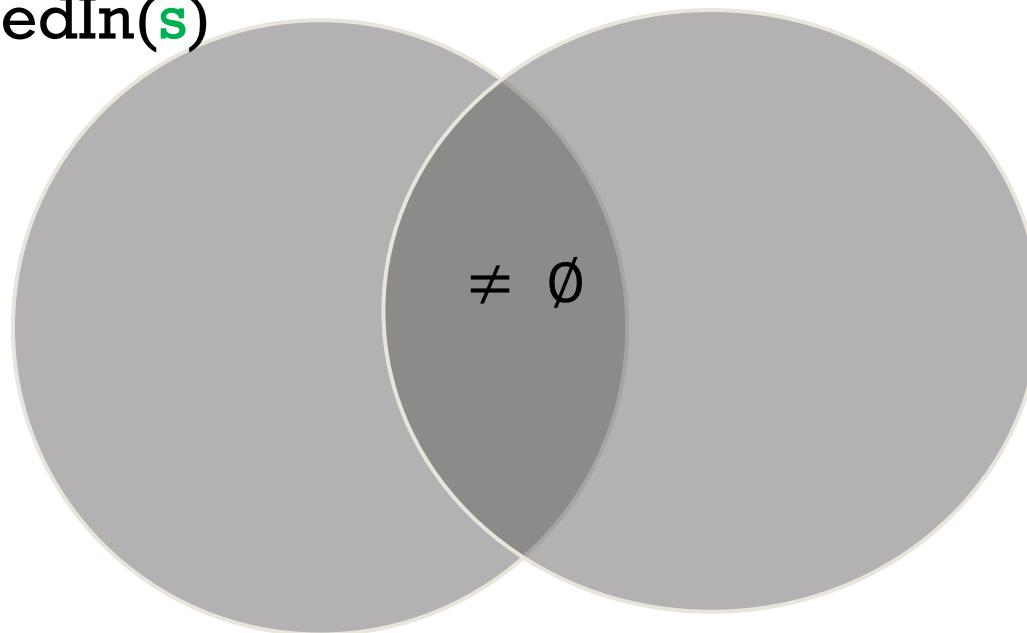


# SOME

Find each  $(s, d)$  pair such that student  $s$  takes **some** courses offered by department  $d$

CoursesEnrolledIn( $s$ )

CoursesOfferedBy( $d$ )



# DEFINING RELEVANT SETS WITH VIEWS

- Definition of CoursesEnrolledIn(sid)

```
CREATE FUNCTION CoursesEnrolledIn(sid TEXT)
RETURNS TABLE (cno TEXT) AS
$$
SELECT E.cno
FROM Enroll E
WHERE E.sid = CoursesEnrolledIn.sid;
$$ LANGUAGE SQL
```



# DEFINING RELEVANT SETS WITH VIEWS

- Definition of CoursesOfferedBy(dept)

```
CREATE FUNCTION Courses OfferedBy(dept TEXT)
    RETURNS TABLE (cno TEXT) AS
$$
    SELECT  C.cno
    FROM    Course C
    WHERE   C.dept = CoursesOfferedBy.dept;
$$ LANGUAGE SQL
```



# SOME FOR Pairs

$$A \cap B \neq \emptyset \quad \exists x (x \in A \cap B)$$

```
SELECT S.sid, D.dept
FROM Student S, Department D
WHERE EXISTS (SELECT cno
               FROM CoursesEnrolledIn(S.sid)
               INTERSECT
               SELECT cno
               FROM CoursesOfferedBy(D.dept))
```



# SOME FOR Pairs

```
SELECT S.sid, D.dept
FROM Student S, Department D
WHERE EXISTS (SELECT cno
               FROM CoursesEnrolledIn(S.sid)
               WHERE
                     cno IN (SELECT cno
                             FROM CoursesOfferedBy(D.dept))
```



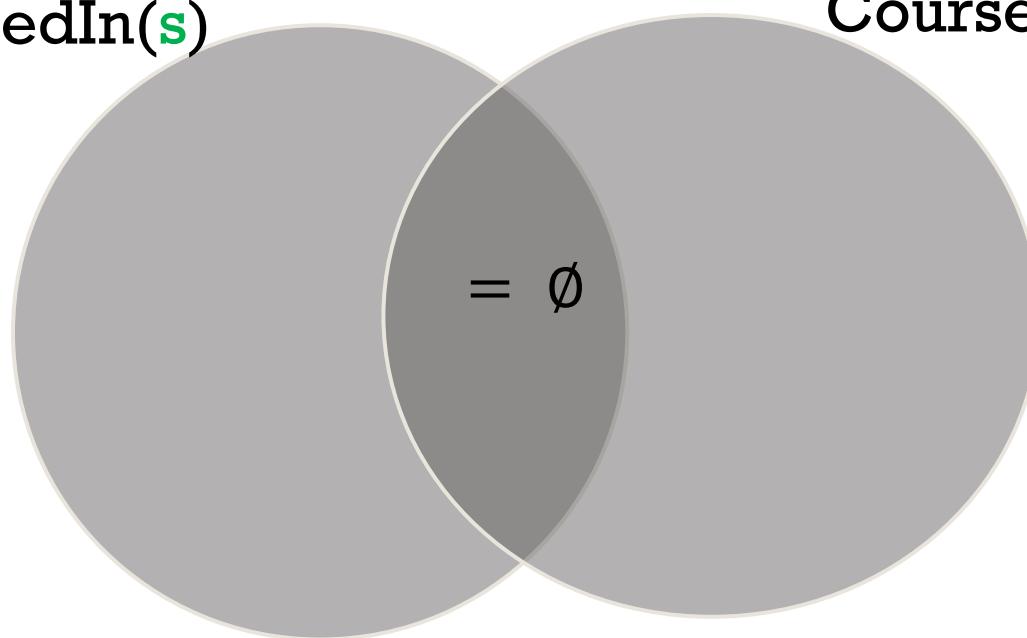
# NO FOR PAIRS

$$A \cap B = \emptyset \quad \neg \exists x (x \in A \cap B)$$

Find each ( $s, d$ ) pair such that student  $s$  takes **no** courses offered by department  $d$

CoursesEnrolledIn( $s$ )

CoursesOfferedBy( $d$ )



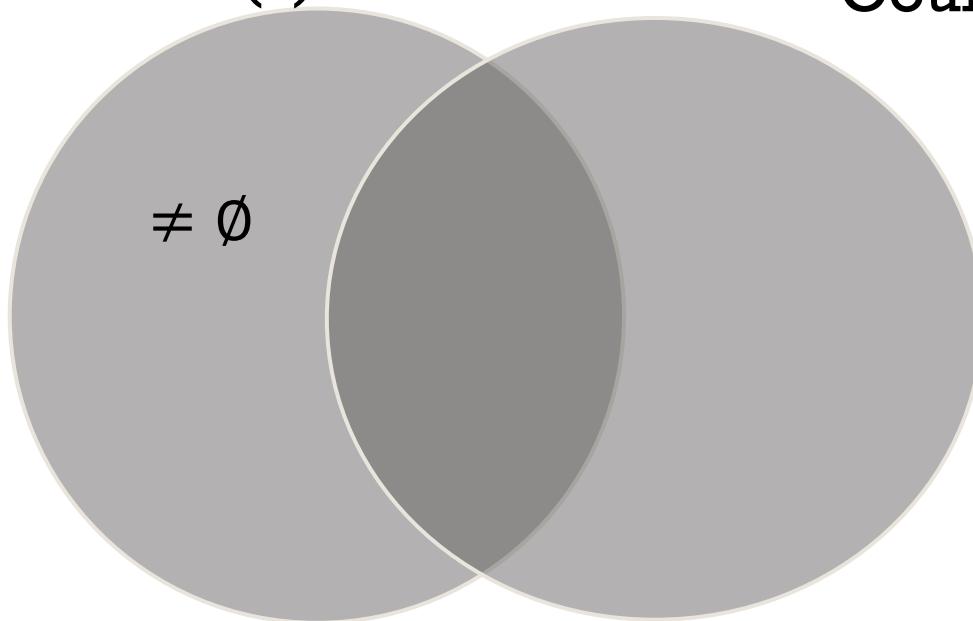
# NOT ONLY FOR PAIRS

$$A - B \neq \emptyset \quad \exists x (x \in A - B)$$

Find each  $(s, d)$  pair such that student  $s$  takes **not only** courses offered by department  $d$

CoursesEnrolledIn( $s$ )

CoursesOfferedBy( $d$ )



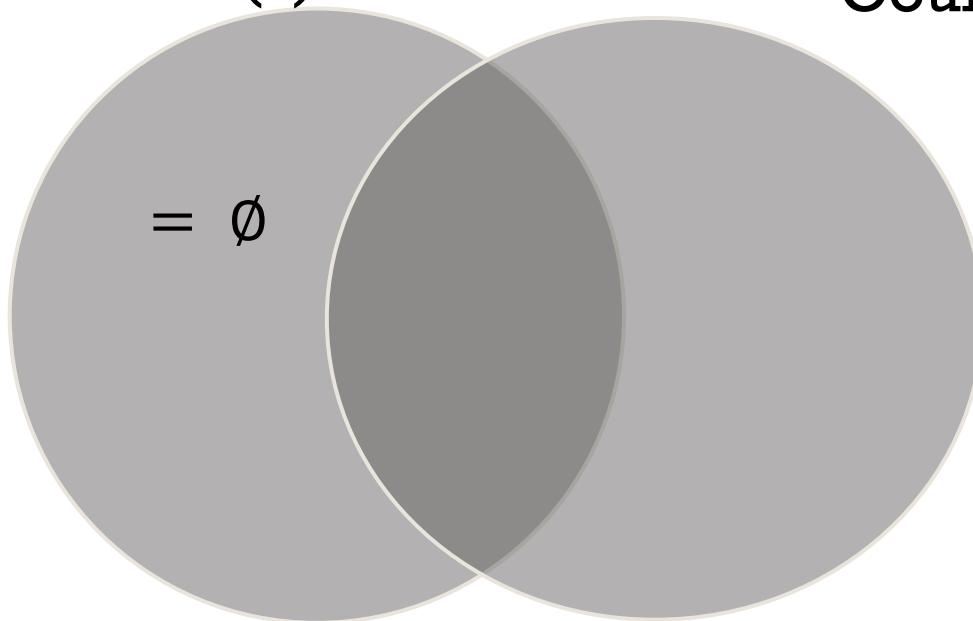
# ONLY FOR PAIRS (SUBSET JOIN)

$$A - B = \emptyset \quad \neg \exists x (x \in A - B)$$

Find each ( $s, d$ ) pair such that student  $s$  takes **only** courses offered by department  $d$

CoursesEnrolledIn( $s$ )

CoursesOfferedBy( $d$ )



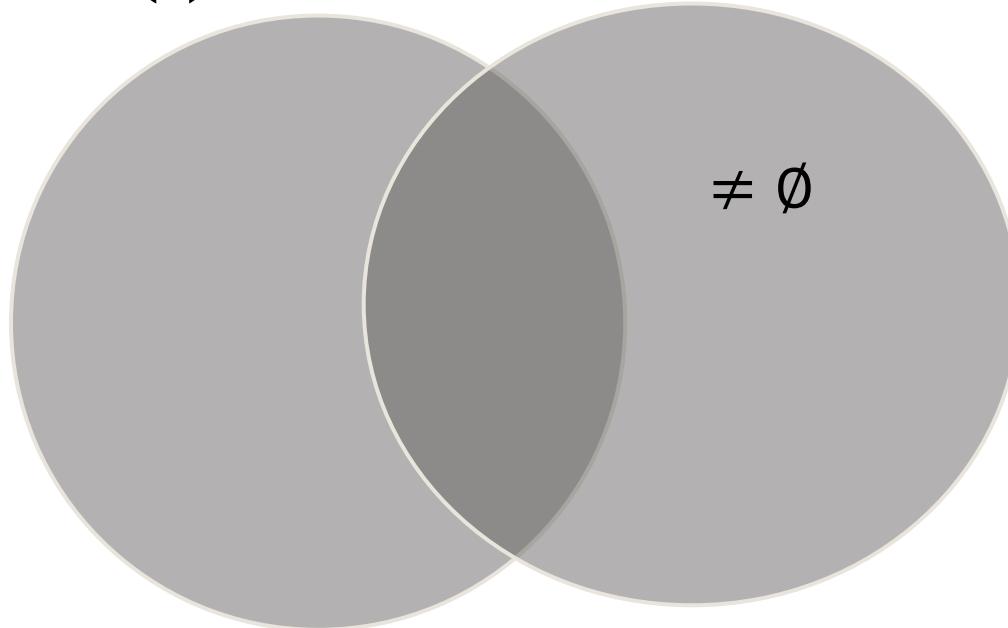
# NOT ALL FOR PAIRS

$$B - A \neq \emptyset \quad \exists x (x \in B - A)$$

Find each (**s,d**) pair such that student **s** takes **not all** courses offered by department **d**

CoursesEnrolledIn(**s**)

CoursesOfferedBy(**d**)



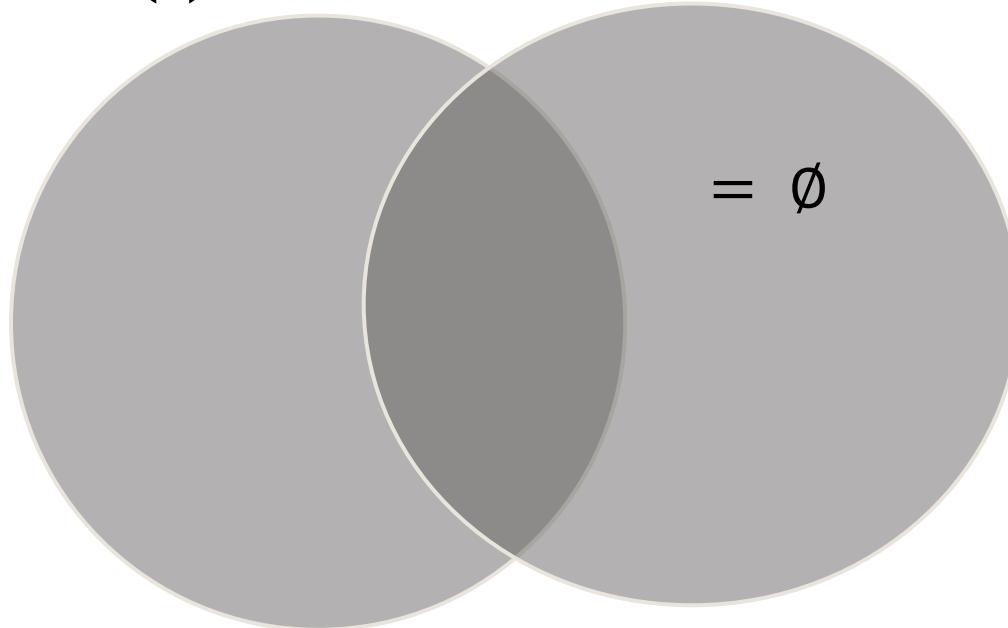
# ALL FOR PAIRS

$$B - A = \emptyset \quad \neg \exists x (x \in B - A)$$

Find each ( $s, d$ ) pair such that student  $s$  takes **all** courses offered by department  $d$

CoursesEnrolledIn( $s$ )

CoursesOfferedBy( $d$ )



# **QUERIES WITH QUANTIFIERS USING COUNT FUNCTION**



# QUERIES WITH QUANTIFIERS

- Find the sid of each student who takes **some** CS courses
- Find the sid of each student who takes **no** CS courses
- Find the sid of each student who takes **not only** CS courses
- Find the sid of each student who takes **only** CS courses
- Find the sid of each student who takes **not all** CS courses
- Find the sid of each student who takes **all** CS courses
- Find the sid of each student who takes **at least half of** CS courses

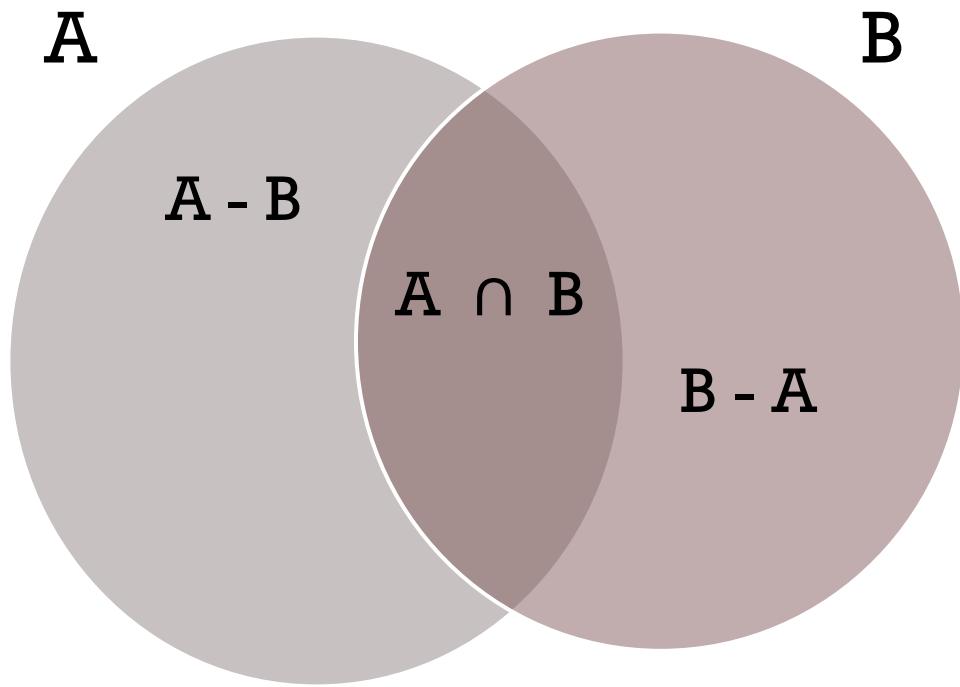


# VENN DIAGRAM AND SQL TEMPLATE

- There is a **Venn diagram with conditions** to express a query with a quantifier
- These conditions can be expressed as **counting conditions**
- There is a corresponding SQL statement to express this Venn diagram with condition using the **COUNT function**



# VENN DIAGRAM OF 2 SETS



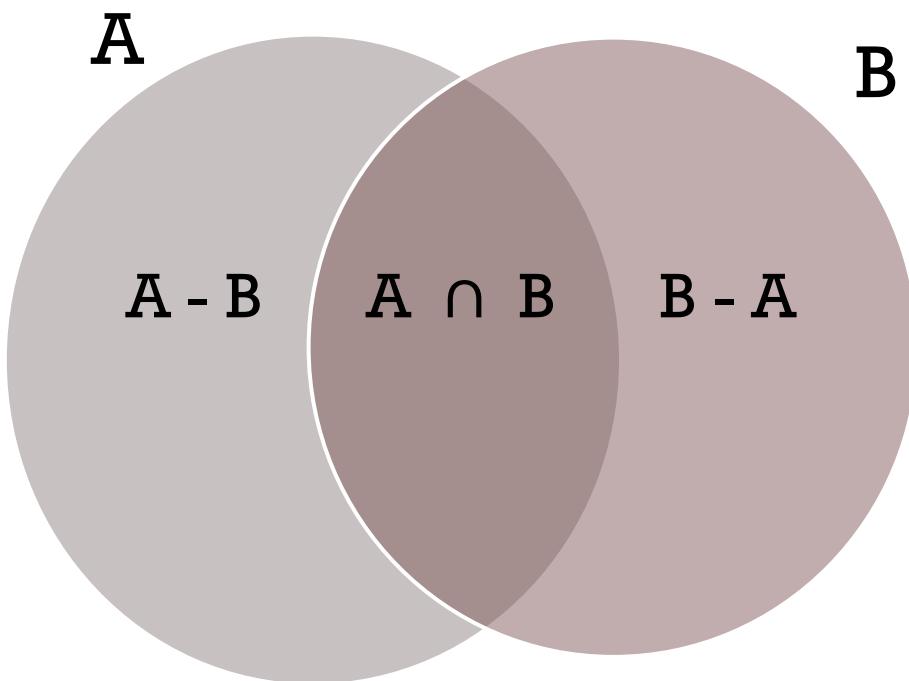
$A - B$  Left Ear

$B - A$  Right Ear

$A \cap B$  Lens



# VENN DIAGRAM OF 2 SETS WITH CONDITIONS



| Condition                                                   |
|-------------------------------------------------------------|
| $A \cap B \neq \emptyset \Leftrightarrow  A \cap B  \geq 1$ |
| $A \cap B = \emptyset \Leftrightarrow  A \cap B  = 0$       |
| $A - B \neq \emptyset \Leftrightarrow  A - B  \geq 1$       |
| $A - B = \emptyset \Leftrightarrow  A - B  = 0$             |
| $B - A \neq \emptyset \Leftrightarrow  B - A  \geq 1$       |
| $B - A = \emptyset \Leftrightarrow  B - A  = 0$             |
| $ A - B  = 0 \text{ and }  B - A  = 0$                      |
| $ A \cap B  \geq 2$                                         |
| ...                                                         |



# VENN DIAGRAM FOR OUR QUERIES

- For a student with key sid,  $\text{CoursesEnrolledIn(sid)}$  denotes the set of courses taken by this student
- $\text{CS_courses}$  denotes the set of courses offered by the 'CS' department
- So in our previous Venn diagram

$A = \text{CoursesEnrolledIn}(sid)$

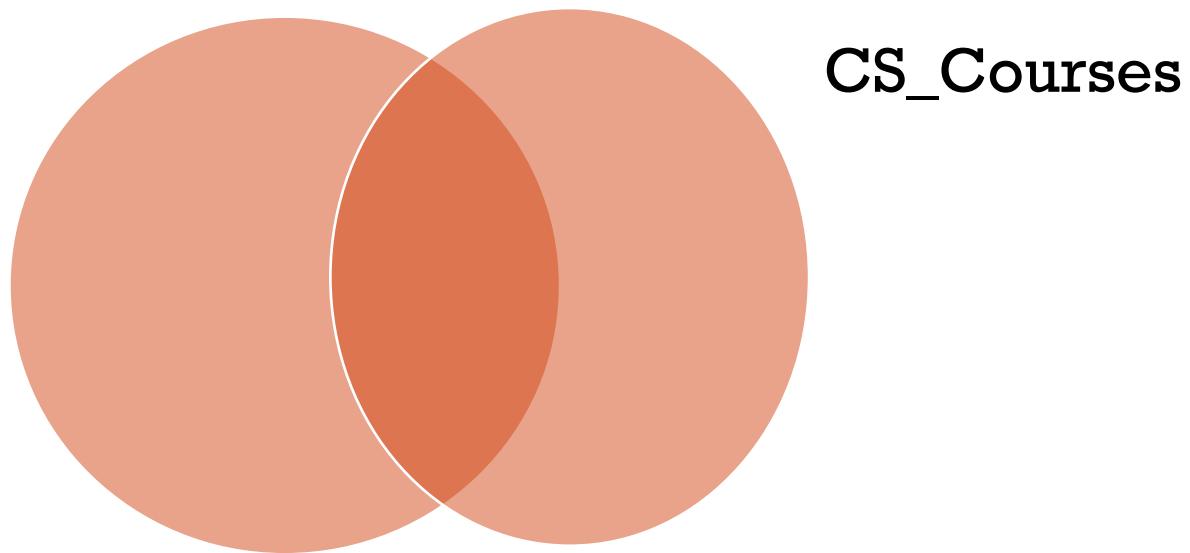
$B = \text{CS_Courses}$

- Note that for different values of sid,  $\text{CoursesEnrolledIn}(sid)$  denote different sets



# VENN DIAGRAM FOR OUR QUERIES

CoursesEnrolledIn(**sid**)

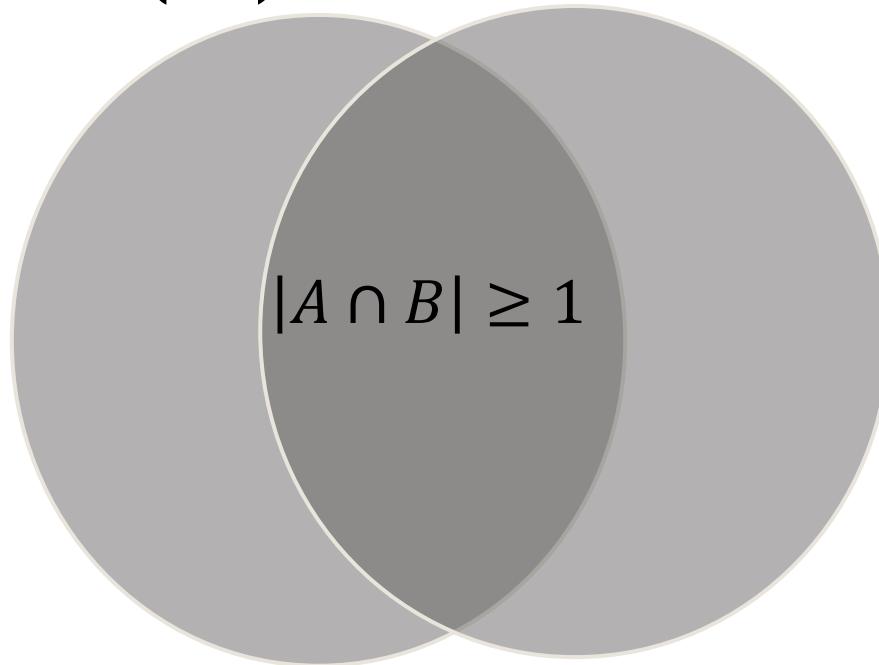


# SOME

Find **sid** of each student who takes **some** CS courses

CoursesEnrolledIn(**sid**)

CS\_courses



# SOME

```
SELECT sid  
FROM Student S  
WHERE (SELECT COUNT(l)  
      FROM (SELECT cno  
            FROM CoursesEnrolledIn(sid)  
            INTERSECT  
            SELECT cno  
            FROM CS_Courses) q) ≥ 1;
```

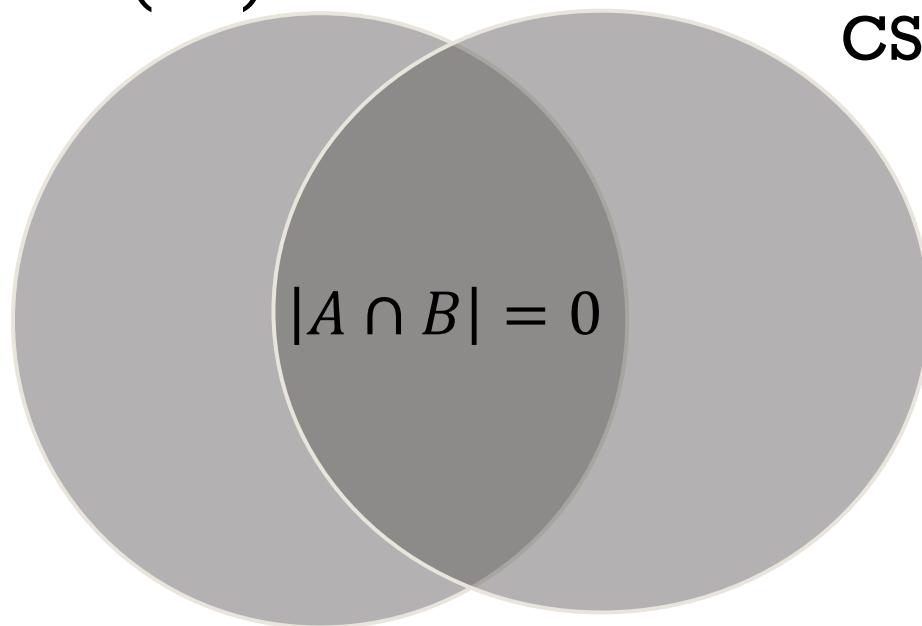


**NO**

Find sid of each student who takes **no** CS courses

CoursesEnrolledIn(sid)

CS courses



# NO

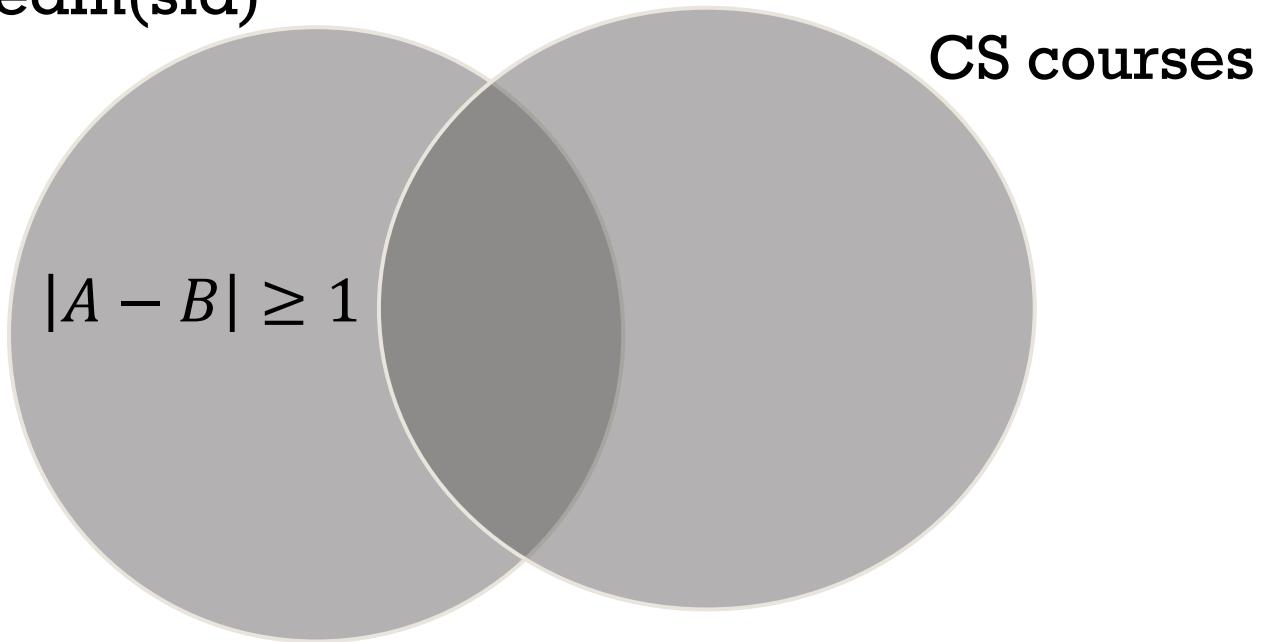
```
SELECT sid  
FROM Student S  
WHERE (SELECT COUNT(l)  
      FROM (SELECT cno  
            FROM CoursesEnrolledIn(sid)  
            INTERSECT  
            SELECT cno  
            FROM CS_Courses) q) = 0;
```



# NOT ONLY

Find sid of each student who takes **not only** CS courses

CoursesEnrolledIn(sid)



# NOT ONLY

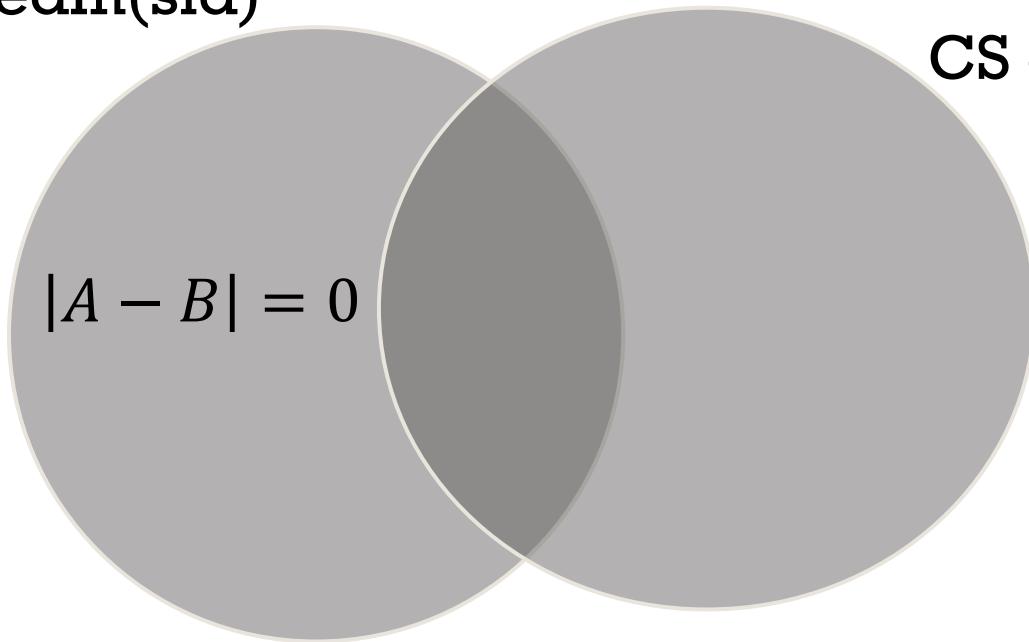
```
SELECT sid  
FROM Student S  
WHERE (SELECT COUNT(l)  
      FROM (SELECT cno  
            FROM CoursesEnrolledIn(sid)  
            EXCEPT  
            SELECT cno  
            FROM CS_Courses) q) ≥ 1;
```



# ONLY

Find sid of each student who takes **only** CS courses

CoursesEnrolledIn(sid)



CS courses

$$|A - B| = 0$$



# ONLY

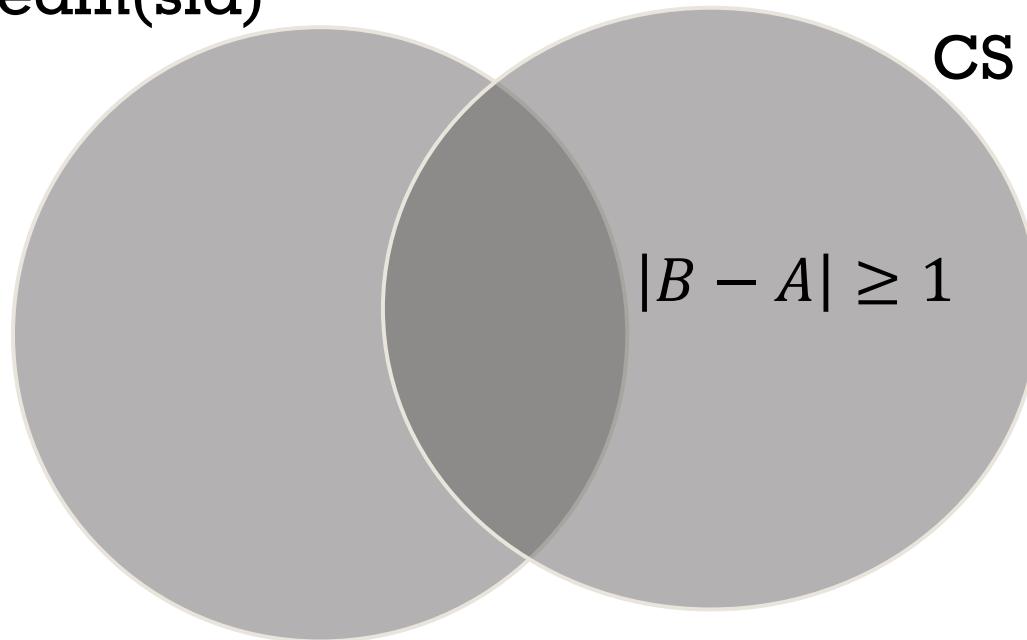
```
SELECT sid  
FROM Student S  
WHERE (SELECT COUNT(l)  
      FROM (SELECT cno  
            FROM CoursesEnrolledIn(sid)  
            EXCEPT  
            SELECT cno  
            FROM CS_Courses) q) = 0;
```



# NOT ALL

Find sid of each student who takes **not all** CS courses

CoursesEnrolledIn(sid)



CS courses

$$|B - A| \geq 1$$



# NOT ALL

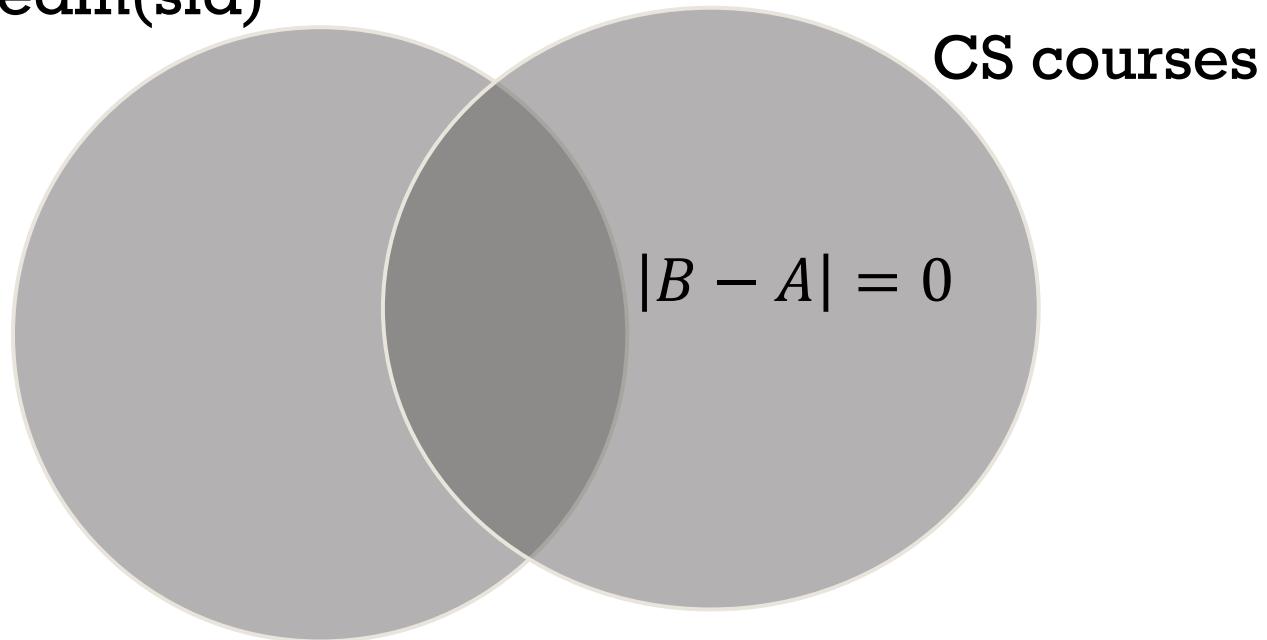
```
SELECT sid  
FROM Student S  
WHERE (SELECT COUNT(1)  
      FROM (SELECT cno  
            FROM CS_courses  
            EXCEPT  
            SELECT cno  
            FROM CoursesEnrolledIn(sid))) q) ≥ 1;
```



# ALL

Find sid of each student who takes **all** CS courses

CoursesEnrolledIn(sid)



# ALL

```
SELECT sid  
FROM Student S  
WHERE (SELECT COUNT(1)  
       FROM (SELECT cno  
              FROM CS_courses  
              EXCEPT  
              SELECT cno  
              FROM CoursesEnrolledIn(sid))) q) = 0;
```



# ALL AND ONLY

We have multiple quantifiers: **ALL** and **ONLY**

These must both be specified in the WHERE clause using the **ALL** and **ONLY** quantifier templates:

```
SELECT S.Sid  
FROM Student S  
WHERE ALL template AND  
ONLY template
```



# AT LEAST TWO

Find **sid** of each student who takes **at least two** CS courses

CoursesEnrolledIn(**sid**)

CS\_courses

$$|A \cap B| \geq 2$$



# AT LEAST 2

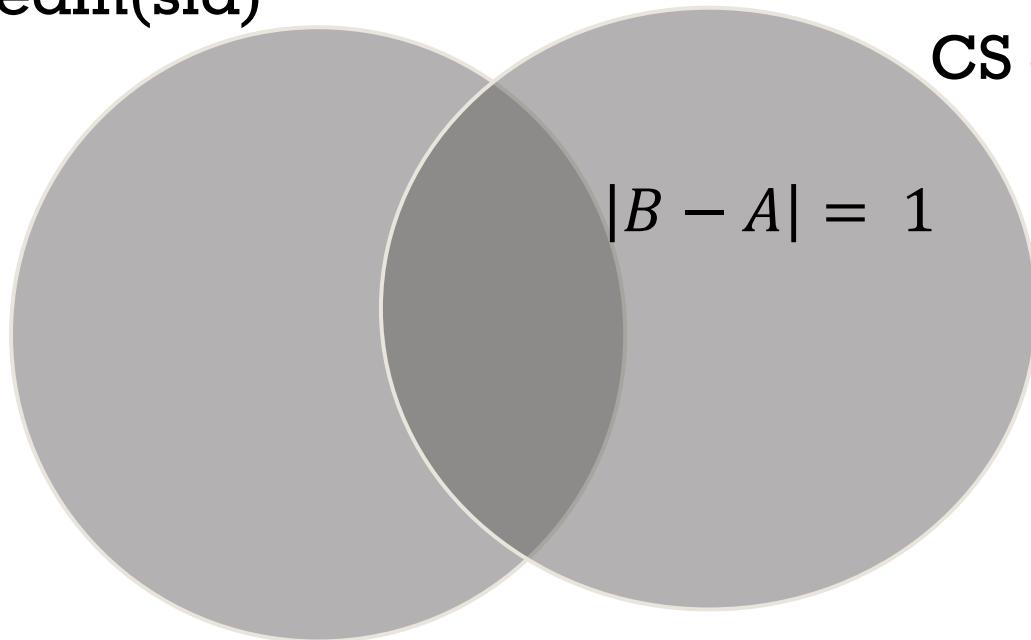
```
SELECT sid  
FROM Student S  
WHERE (SELECT COUNT(l)  
      FROM (SELECT cno  
            FROM CoursesEnrolledIn(sid)  
            INTERSECT  
            SELECT cno  
            FROM CS_Courses) q) ≥ 2;
```



# ALL BUT ONE

Find sid of each student who takes **all but one** CS courses

CoursesEnrolledIn(sid)



# ALL BUT ONE

```
SELECT sid  
FROM Student S  
WHERE (SELECT COUNT(1)  
       FROM (SELECT cno  
              FROM CS_courses  
              EXCEPT  
              SELECT cno  
              FROM CoursesEnrolledIn(sid))) q) = 1;
```

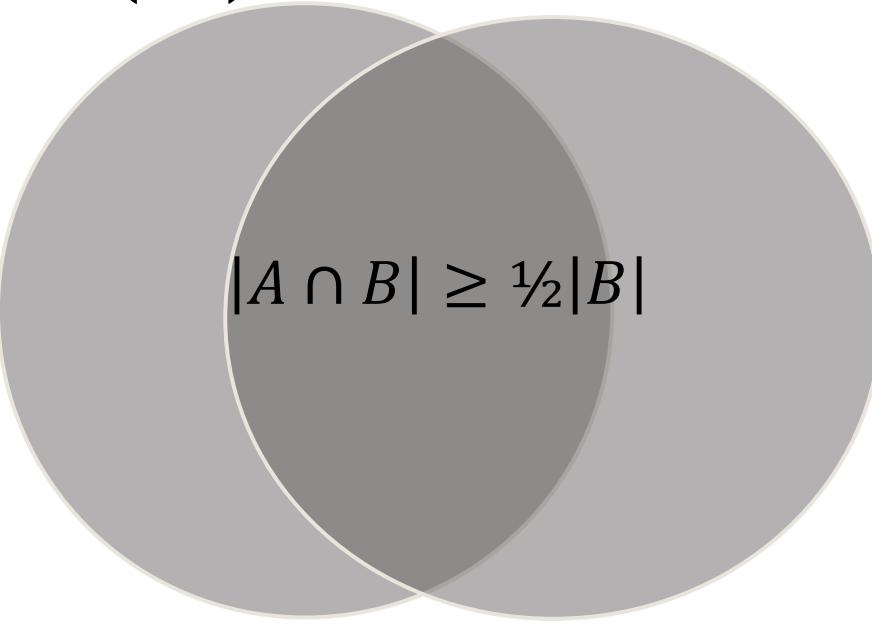


# AT LEAST HALF OF

Find the sid of each student who takes **at least half of** the CS courses

CoursesEnrolledIn(sid)

CS courses


$$|A \cap B| \geq \frac{1}{2}|B|$$



# AT LEAST HALF OF

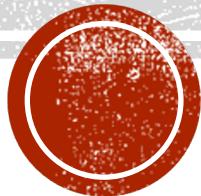
```
SELECT sid
FROM Student S
WHERE 2* (SELECT COUNT(1)
           FROM (SELECT cno
                  FROM CoursesEnrolledIn(sid)
                INTERSECT
                  SELECT cno
                  FROM CS_Courses) q) ≥ (SELECT
   COUNT (1)
   FROM CS_courses)

|A ∩ B| ≥ ½|B|
2 |A ∩ B| ≥ |B|
```



# TRIGGERS

Dirk Van Gucht



# TRIGGERS

- A trigger is a user-defined function that is triggered (invoked) when state-changes are applied to a relation or a view with which the trigger is associated
- State changes can be INSERT, DELETE, or UPDATE statements
- Triggers can be applied in different temporal modes:

BEFORE

AFTER

INSTEAD

- Triggers can be applied by **row level** or by single **statement level**
- We will be mainly concerned with row level triggers



# APPLICATIONS OF TRIGGERS

- Materialized view
- View updates
- Constraint verification
- Dynamic aggregate function values maintenance
- Logging/auditing



# TRIGGER STRUCTURE

- The programming structure consists of two components:
  - (1) A **trigger function**
  - (2) A **trigger definition**



# TRIGGER FUNCTION

- A trigger function is created with the CREATE FUNCTION command, declaring it as a function with no arguments and a return type of trigger.
- The function must be declared with no arguments even if it expects to receive arguments specified in CREATE TRIGGER
- When a function is called as a trigger, several special variables are created automatically. Two important variables are:
  - NEW: variable holding the new database row for INSERT/UPDATE operations in row-level triggers.
  - OLD: variable holding the old database row for UPDATE/DELETE operations in row-level triggers.



# EXAMPLE: *INSERT INTO TRIGGER* ON A RELATION

- We will consider an **INSERT** trigger on the relation **Student(sid, sname, major, byear)**
- We assume there is a relation **CS\_student(sid,sname,byear)** which holds the information about 'CS' majors
- Objective: an **INSERT INTO Student** should trigger an **INSERT INTO the CS\_student relation** if the student is a 'CS' major



# EXAMPLE: INSERT INTO TRIGGER ON A RELATION

- Trigger function (describes trigger action to be done)

```
CREATE OR REPLACE FUNCTION insert_into_Student() RETURNS TRIGGER AS
$$
BEGIN
  IF NEW.Major = 'CS' THEN
    INSERT INTO CS_Student VALUES (NEW.sid, NEW.sname, NEW.byear);
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';
```

- The variable **NEW** is a system-provided variable that refers to the tuple that is inserted into Student relation



# EXAMPLE: INSERT INTO TRIGGER ON A RELATION

Trigger definition (declaration) on the Student relation

```
CREATE TRIGGER insert_into_Student_Relation  
    BEFORE INSERT ON Student  
    FOR EACH ROW  
    EXECUTE PROCEDURE insert_into_Student();
```

**BEFORE INSERT** tells us that an action needs to be executed before the tuple is inserted into Student

That action is specified by the code in the trigger function `insert_into_Student()`



# EFFECTS

**INSERT INTO Student VALUES ('s5', 'Linda', 'Math', 1960)**

will insert this tuple only in the Student relation

**INSERT INTO Student VALUES ('s6', 'Marc', 'CS', 1963)**

will insert the tuple ('s6', 'Marc', 1963) in the CS\_course relation first (BEFORE) and then it

will insert this tuple ('s6', 'Marc', 'CS', 1963) in the Student relation



# FOR EACH ROW CLAUSE

- The FOR EACH CLAUSE guarantees that the trigger function is applied for each possible tuple that can be assigned to the NEW variable



# FOR EACH ROW SEMANTICS

- Multiple inserts will be handled one at a time

```
INSERT INTO Student VALUES ('s7', 'Nick', 'CS', 1990),  
                           ('s8', 'Vince', 'Biology', 1985);
```

- The NEW variable will be assigned to the first of these tuples
  - The trigger function will work on that NEW variable
  - Subsequently, the NEW variable will be assigned to the second of these tuples
  - The trigger function will then work on this NEW variable
- 
- Both tuples will be inserted into the Student relation
  - But only the tuple ('s7', 'Nick', 1990) will be inserted into the CS\_student relation



# RETURNING NEW VERSUS NULL

- Consider a small change to the trigger function
- **RETURN NULL** instead of **RETURN NEW**

```
CREATE OR REPLACE FUNCTION insert_into_Student() RETURNS
TRIGGER AS
$$
BEGIN
IF NEW.Major = 'CS' THEN
    INSERT INTO CS_Student VALUES (NEW.sid, NEW.sname,
NEW.byear);
END IF;
RETURN NULL;
END;
```

- **\$\$ LANGUAGE 'plpgsql';**
- The insertion of a tuple into Student **will be ignored**
- The insertion of a tuple in CS\_student **will still be done**



# BEFORE VERSUS AFTER

- With **AFTER** trigger on an insert into Student, the insert will always succeed before the trigger function executes
- In the **AFTER** case, a RETURN NEW statement will do nothing.
- So it is typical to return **NULL** for AFTER triggers



# AFTER TRIGGER

```
CREATE OR REPLACE FUNCTION insert_into_Student() RETURNS TRIGGER
AS
$$
BEGIN
IF NEW.Major = 'CS' THEN
    INSERT INTO CS_Student VALUES (NEW.sid, NEW.sname NEW.byear);
END IF;
RETURN NULL; -- Equivalent with RETURN NEW
END;
$$ LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER insert_into_Student_Relation
AFTER INSERT ON Student
FOR EACH ROW
EXECUTE PROCEDURE insert_into_Student();
```



# DELETE TRIGGER (**OLD** VARIABLE)

```
CREATE OR REPLACE FUNCTION delete_from_Student()
RETURNS TRIGGER AS
$$
BEGIN
IF OLD.Major = 'CS' THEN
    DELETE FROM CS_student WHERE sid = OLD.sid;
END IF;
RETURN OLD;
END;
$$ LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER delete_from_Student_Relation
BEFORE DELETE ON Student
FOR EACH ROW
EXECUTE PROCEDURE delete_from_Student();
```



# FOR EACH ROW CLAUSE

- The FOR EACH CLAUSE guarantees that the trigger function is applied for each possible tuple that can be assigned to the OLD variable



# VIEW UPDATE (INSERT)

- Base relation Student(Sid, Sname, Major,Byear)
- Consider a **view** for CS\_student

```
CREATE VIEW CS_Student AS  
SELECT sid, sname, byear  
FROM Student  
WHERE Major = 'CS'
```

- Consider the insertion of new CS students into the CS\_student view
- The desired effect is to insert these new students into the base relation Student and specify their major as 'CS'



# **VIEW UPDATE (INSERT)**

**INSERT INTO CS\_Student VALUES ('s1', 'Eric', 1990);**

should “trigger”

**INSERT INTO Student VALUES('s1', 'Eric', 'CS', 1990);**

**Remark:** Notice that we can always insert into the Student relation



# TRIGGER FUNCTION DEFINITION

- We first define a trigger function that accomplishes each insert into the base relation Student

```
CREATE FUNCTION insert_CS_student() RETURNS TRIGGER AS
$$
BEGIN
    INSERT INTO Student VALUES(NEW.sid, NEW.sname, 'CS', NEW.byear);
    RETURN NULL;
END;
$$ LANGUAGE 'plpgsql';
```



# TRIGGER DEFINITION FOR INSERT INTO VIEW

- After the trigger function has been defined, we bind it to a trigger declared for the CS\_Student view

```
CREATE TRIGGER insert_into_CS_student
INSTEAD OF INSERT ON CS_Student
FOR EACH ROW
EXECUTE PROCEDURE insert_CS_Student();
```

- Notice that we must use INSTEAD



# VIEW UPDATE (DELETE)

- Consider the deletion of students from the CS\_student view
- The desired effect is to delete those students from the base relation Student

```
CREATE OR REPLACE FUNCTION delete_CS_student() RETURNS TRIGGER AS
$$
BEGIN
    DELETE FROM Student WHERE sid = OLD.sid;
    RETURN NULL;
END;
$$ LANGUAGE 'plpgsql';
```



# TRIGGER FUNCTION AND TRIGGER DEFINITIONS FOR DELETION FROM VIEW

- Trigger function

```
CREATE FUNCTION delete_CS_student() RETURNS TRIGGER AS
$$
BEGIN
    DELETE FROM Student WHERE sid = OLD.sid;
    RETURN NULL;
END;
$$ LANGUAGE 'plpgsql';
```

- Trigger

```
CREATE TRIGGER delete_from_CS_student
INSTEAD OF DELETE ON CS_Student
FOR EACH ROW
EXECUTE PROCEDURE delete_CS_Student();
```



# VIEW UPDATE (DELETE)

- **OLD** is a system-maintained tuple variable that refers to any tuple that is to be deleted from the CS\_Student view

```
DELETE FROM CS_Student WHERE sname = 'John';
```

This should trigger deleting all students with name 'John' and major 'CS' from the Student table

```
DELETE FROM Student WHERE sname = 'John' AND  
                      major = 'CS';
```



# VIEW MATERIALIZATION

- Consider maintaining a relation Math\_Student that is the materialization of a view for Math students

```
CREATE TABLE Math_Student  
(Sid INTEGER,  
Sname TEXT);
```



# MATERIALIZATION: TRIGGER FUNCTION AND TRIGGER DEFINITIONS

```
CREATE FUNCTION insert_Math_student() RETURNS TRIGGER AS
$$
BEGIN
  IF (NEW.Major = 'Math')
    THEN INSERT INTO Math_Student VALUES(NEW.sid, NEW.sname);
  END IF;
  RETURN NULL;
END;
$$ LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER add_Math_Student
AFTER INSERT ON Student
FOR EACH ROW
EXECUTE PROCEDURE insert_Math_Student();
```



# CONSTRAINT VERIFICATION (KEYS)

- Ensure that sid is a primary key

```
CREATE OR REPLACE FUNCTION check_Student_key_constraint() RETURNS trigger AS
$$
BEGIN
    IF NEW.sid IN (SELECT sid FROM Student) THEN
        RAISE EXCEPTION 'sid already exists';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER check_Student_key_
BEFORE INSERT
ON Student
FOR EACH ROW
EXECUTE PROCEDURE check_Student_key_constraint();
```



# AGGREGATE FUNCTION MAINTENANCE

- Keep track of number of students

```
CREATE TABLE Count_Students( total integer);
INSERT INTO  Count_Students VALUES(0);
```

```
CREATE OR REPLACE FUNCTION Maintain_Number_Students() RETURNS trigger AS
$$
BEGIN
    UPDATE Count_Students SET total = total + 1;
    RETURN NULL;
END;
$$ LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER Total_Students
AFTER INSERT ON Student
FOR EACH ROW
EXECUTE PROCEDURE Maintain_Number_Students();
```



# ALTERNATIVE SOLUTION

```
CREATE OR REPLACE FUNCTION Maintain_Number_Students() RETURNS trigger AS
$$
BEGIN
    UPDATE Count_Students SET total = (SELECT COUNT(*) FROM Student);
    RETURN NULL;
END;
$$ LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER Total_Students
AFTER INSERT ON Student
EXECUTE PROCEDURE Maintain_Number_Students();
```

- In this case, Total\_Students is a **STATEMENT trigger**:  
the trigger function is executed once independent of  
the number of tuples inserted in the Student relation



# MAINTAINING A LOG

- Any time we insert a new student, we want to **log** a record with the **time stamp** for that insert
- We will maintain a `Student_log` relation for this

```
CREATE TABLE Student_log(  
    sid text,  
    stamp timestamp  
)
```



# MAINTAINING A LOG

- We can do this with the following trigger

```
CREATE OR REPLACE FUNCTION time_stamp_for_student() RETURNS
TRIGGER AS
```

```
$$
BEGIN
    INSERT INTO Student_log VALUES (NEW.sid, now());
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER log_student
AFTER INSERT ON Student
FOR EACH ROW EXECUTE PROCEDURE time_stamp_for_student();
```



# TRIGGERS CAN LEAD TO INFINITE LOOPS

```
CREATE TABLE A (x INTEGER);
```

```
CREATE FUNCTION insert_in_A () RETURNS TRIGGER AS
$$
BEGIN
    INSERT INTO A VALUES(NEW.x);
    RETURN NEW;
END;
$$ LANGUAGE 'plpgsql';
```

```
CREATE TRIGGER insert_into_A_relation
BEFORE INSERT ON A
FOR EACH ROW EXECUTE PROCEDURE insert_in_A();
```

```
INSERT INTO A VALUES(1);
```

This will lead to an infinite loop;



# OTHER ISSUES

- Triggers can be **hard to debug**
- When multiple triggers are defined on the same relation, the **order** in which these triggers get **evaluated** becomes relevant
- Trigger can incur **performance overhead**
- Triggers are **powerful aids in managing and protecting the state** of the database in sophisticated ways
- Triggers are a fundamental component of **event-driven programming**



# The Relational Algebra

Dirk Van Gucht<sup>1</sup>

<sup>1</sup>Indiana University

February 5, 2019

## Outline

- Motivation
- Syntax of Relational Algebra (RA)
- Semantics of RA in SQL
- Expressing queries in RA

## Relational Algebra (Motivation)

- SQL is a declarative language to specify queries over relational databases
- RA is a procedural language wherein algebraic expressions specify queries
- RA expressions provide procedures (algorithms) to evaluate queries
  - Examining these procedure provides insights about the time and space complexities of evaluating queries

## Relational Algebra (Motivation)

- SQL<sup>1</sup> and RA express the same queries:
  - Each SQL query can be expressed by an RA expression
  - Each RA expression can be expressed by a SQL query.
- **Query optimization:** Rewrite rules can transform a RA expression into another equivalent RA expression that is more efficient to evaluate
- **Query evaluation:** Different algorithms and data structures can be associated with RA operations for efficient evaluations

---

<sup>1</sup>without aggregation functions

## Syntax of RA

- The relational algebra is a **typed** language of expressions
- Starting from relations and constants, RA expressions are inductively built using the operators

| Operator          | Algebraic notation  |
|-------------------|---------------------|
| cartesian product | $\times$            |
| selection         | $\sigma(\cdot)$     |
| projection        | $\pi \dots (\cdot)$ |
| union             | $\cup$              |
| intersection      | $\cap$              |
| difference        | $-$                 |

- Each RA expression has a **schema** which is its type
- We will use  $E(A_1, \dots, A_m)$  to denote a RA expression  $E$  with attribute schema  $(A_1, \dots, A_m)$

## RA basic expressions: relations

- Let  $R(A_1, \dots, A_m)$  be a relation
- Then  $R$  is a RA expression with schema  $(A_1, \dots, A_m)$
- Its value is the set of tuples from  $R$ , i.e., the relation instance associated with  $R$
- $R$  can be expressed by SQL query

```
SELECT r.A1, ..., r.Am
FROM   R r
```

## RA basic expressions: constants

- Let  $A$  be an attribute and let  $a$  be a constant
- Then  $(A:a)$  is a RA expression with schema  $(A)$
- Its value is a unary relation containing the single tuple  $(a)$
- $(A : a)$  can be expressed by SQL query

SELECT a AS A

|   |
|---|
| A |
| a |

## RA operators: the cartesian product $\times$

- Let  $R(A_1, \dots, A_m)$  and  $S(B_1, \dots, B_n)$  be two relations with **non-overlapping** schemas
- Then  $R \times S$  is a RA expression with schema  $(A_1, \dots, A_m, B_1, \dots, B_n)$
- Its value is the set of all tuples that can be obtained by **pairing** each tuple  $r$  of  $R$  with each tuple  $s$  of  $S$
- $R \times S$  can be expressed by the SQL query

```
SELECT r.A1, ..., r.Am, s.B1, ..., s.Bn
  FROM Rr, Ss
```

- Observe that  $|R \times S| = |R||S|$

## RA operators: the cartesian product $\times$ (example)

| R     |       |       |
|-------|-------|-------|
| A     | B     | C     |
| $a_1$ | $b_1$ | $c_1$ |
| $a_2$ | $b_2$ | $c_2$ |
| $a_3$ | $b_3$ | $c_3$ |

| S |  |
|---|--|
| D |  |
| 1 |  |
| 2 |  |

| $R \times S$ |       |       |   |
|--------------|-------|-------|---|
| A            | B     | C     | D |
| $a_1$        | $b_1$ | $c_1$ | 1 |
| $a_2$        | $b_2$ | $c_2$ | 1 |
| $a_3$        | $b_3$ | $c_3$ | 1 |
| $a_1$        | $b_1$ | $c_1$ | 2 |
| $a_2$        | $b_2$ | $c_2$ | 2 |
| $a_3$        | $b_3$ | $c_3$ | 2 |

## RA operators: the cartesian product $\times$ (building search space)

- $R \times S$  provides in **single** relation all the information that is present in  $R$  and  $S$
- $R \times S$  can be interpreted as the **search space**
- It provides a mechanism to associate each tuple  $r$  of  $R$  with each tuple  $s$  of  $S$  into a tuple  $(r, s)$  whose components can be compared
  - I.e., the components  $r.A_1, \dots, r.A_m$  of  $r$  can be compared with the components  $s.B_1, \dots, s.B_n$  of  $s$

## RA operators: selection $\sigma_{A_i \theta a}$ (Introduction)

- Let  $R(A_1, \dots, A_m)$  be a relation
- Let  $a$  be a constant value
- Let  $\theta$  be one of the comparison operators  $=, \neq, <, \leq, >, \geq$
- Then  $\sigma_{A_i \theta a}(R)$  is a RA expression with schema  $(A_1, \dots, A_m)$
- Its value consists of the tuples  $r$  in  $R$  such that  $r.A_i \theta a$  is true
- $\sigma_{A_i \theta a}(R)$  is expressed in SQL by

```
SELECT r.A1, ..., r.Am
FROM   R r
WHERE  r.Ai θ a
```

- Observe that  $|\sigma_{A_i \theta a}(R)| \leq |R|$

## RA operators: selection $\sigma_{A \theta a}$ (Example)

*R*

| A     | B     | C     |
|-------|-------|-------|
| $a_1$ | $b_1$ | $c_1$ |
| $a_2$ | $b_2$ | $c_2$ |
| $a_2$ | $b_1$ | $c_2$ |
| $a_3$ | $b_3$ | $c_2$ |

$\sigma_{B=b_1}(R)$

| A     | B     | C     |
|-------|-------|-------|
| $a_1$ | $b_1$ | $c_1$ |
| $a_2$ | $b_1$ | $c_2$ |

$\sigma_{A \neq a_3}(R)$

| A     | B     | C     |
|-------|-------|-------|
| $a_1$ | $b_1$ | $c_1$ |
| $a_2$ | $b_2$ | $c_2$ |
| $a_2$ | $b_1$ | $c_2$ |

## RA operators: selection $\sigma_{A_i \theta A_j}$ (Introduction)

- Let  $R(A_1, \dots, A_m)$  be a relation
- Let  $\theta$  be one of the comparison operators  $=, \neq, <, \leq, >, \geq$
- Then  $\sigma_{A_i \theta A_j}(R)$  is a RA expression with schema  $(A_1, \dots, A_m)$
- Its value is the set of tuples  $r$  of  $R$  such that  $r.A_i \theta r.A_j$  is true
- $\sigma_{A_i \theta A_j}(R)$  is expressed in SQL by

```
SELECT  r.A1, ..., r.Am
FROM    R r
WHERE   r.Ai θ r.Aj
```

- Observe that  $|\sigma_{A_i \theta A_j}(R)| \leq |R|$

## RA operators: selection $\sigma_{A=B}$ (Example)

*R*

| A                     | B                     | C                     |
|-----------------------|-----------------------|-----------------------|
| <i>a</i> <sub>1</sub> | <i>b</i> <sub>1</sub> | <i>a</i> <sub>1</sub> |
| <i>a</i> <sub>2</sub> | <i>b</i> <sub>2</sub> | <i>c</i> <sub>2</sub> |
| <i>a</i> <sub>2</sub> | <i>b</i> <sub>3</sub> | <i>a</i> <sub>2</sub> |

$\sigma_{A=C}(R)$

| A                     | B                     | C                     |
|-----------------------|-----------------------|-----------------------|
| <i>a</i> <sub>1</sub> | <i>b</i> <sub>1</sub> | <i>a</i> <sub>1</sub> |
| <i>a</i> <sub>2</sub> | <i>b</i> <sub>3</sub> | <i>a</i> <sub>2</sub> |

$\sigma_{A \neq C}(R)$

| A                     | B                     | C                     |
|-----------------------|-----------------------|-----------------------|
| <i>a</i> <sub>2</sub> | <i>b</i> <sub>2</sub> | <i>c</i> <sub>2</sub> |

## RA operators: selection $\sigma$

- A selection acts as a **filter** on  $R$
- It provides a **horizontal slice** of  $R$

## RA operators: projection $\pi$ (Introduction)

- Let  $R(A_1, \dots, A_m)$  be a relation and let  $(B_1, \dots, B_k)$  be a non-empty list of  $k$  attributes of  $R$
- Then  $\pi_{B_1, \dots, B_k}(R)$  is a RA expression with schema  $(B_1, \dots, B_k)$
- Its value is the relation  $\{(r.B_1, \dots, r.B_k) \mid r \in R\}$
- In other words, each tuple  $r$  of  $R$  is "projected" on its  $(B_1, \dots, B_k)$  components.
- $\pi_{B_1, \dots, B_k}(R)$  is expressed in SQL by

```
SELECT DISTINCT    r.B1, ..., r.Bk
  FROM              R r
```

- Observe that  $|\pi_{B_1, \dots, B_k}(R)| \leq |R|$

## RA operators: projection $\pi$ (Example)

*R*

| A     | B     | C     |
|-------|-------|-------|
| $a_1$ | $b_1$ | $c_1$ |
| $a_2$ | $b_2$ | $c_2$ |
| $a_1$ | $b_3$ | $c_1$ |
| $a_2$ | $b_2$ | $c_1$ |

$\pi_{A,C}(R)$

| A     | C     |
|-------|-------|
| $a_1$ | $c_1$ |
| $a_2$ | $c_2$ |
| $a_2$ | $c_1$ |

$\pi_{B,A}(R)$

| B     | A     |
|-------|-------|
| $b_1$ | $a_1$ |
| $b_2$ | $a_2$ |
| $b_3$ | $a_1$ |

Notice that duplicates are eliminated and that projection permits attributes to be permuted

## RA operators: projection $\pi$

- A projection provides a mechanism to get certain columns from  $R$
- It provides a **vertical slice** of  $R$

## Example query in RA and SQL

- Consider the relations Student(sid,sname,age) and Enroll(sno,cno)<sup>4</sup>
- "Find the sid and age of each enrolled student whose name is Ann."
- This query can be expressed by the RA expression  
$$\pi_{\text{sid}, \text{age}}(\sigma_{\text{sid}=\text{sno}}(\sigma_{\text{sname}=\text{'Ann'}}(\text{Student} \times \text{Enroll})))$$

- In SQL

```
SELECT DISTINCT s.sid, s.age
FROM Student s, Enroll e
WHERE s.sname = 'Ann' AND s.sid = e.sno
```

---

<sup>4</sup>Notice that we have used two different names sid and sno to refer uniquely to a student. We did this to ensure that the relations have no commonly named attributes.

## Boolean conditions in selection operations

- It is possible to extend the selection operations by permitting boolean combinations of basic conditions " $A_j \theta a$ " and " $A_i \theta A_j$ " using the boolean connectors  $\wedge$ ,  $\vee$ , and  $\neg$ .
- "Find the sid and age of each enrolled student whose name is Ann."
- This query can then be expressed using the following RA expression

$$\pi_{\text{sid}, \text{age}}(\sigma_{\text{sname} = \text{'Ann'}} \wedge \text{sid} = \text{sno}(\text{Student} \times \text{Enroll}))$$

- Notice that the structure of this query is nearly the same as that of its SQL equivalent

## Some comments about attribute names (renaming)

- Cartesian products require that the attributes names of the participating expressions do not overlap
- There are various ways to overcome this by renaming attributes or using some conventions
- For example, let  $R(A, B)$  and  $S(B, C)$  be two relations. We can not write  $R \times S$  since attribute  $B$  occurs in the schemas of  $R$  and  $S$ . Nonetheless we will permit this and agree that the output schema of  $R \times S$  is  $(A, R.B, S.B, C)$  where we have used the relation names  $R$  and  $S$  to differentiate the  $B$  attribute in  $R$  from the  $B$  attribute in  $S$

## Some comments about attribute names

- We can also not write  $R \times R$ . To overcome this problem, we assume that for each relation name  $R$ , there is a series of relations  $R_1, R_2, \dots$  that are "copies" of  $R$ .
- Then instead of writing  $R \times R$ , we write  $R_1 \times R_2$  and the output schema will be  $(R_1.A, R_1.B, R_2.A, R_2.B)$
- In SQL, we can always use the **AS** clause to deal with attribute renaming. We will not give the details of attribute renaming in general but assume that it can always be done appropriately.

## Some comments about attribute names (Example)

- "Find the sids of students who take at least two courses."

$$\pi_{E1.sid}(\sigma_{E1.sid = E2.sid \wedge E1.cno \neq E2.cno}(Enroll_1 \times Enroll_2))$$

where  $E_1$  and  $E_2$  are abbreviations for  $Enroll_1$  and  $Enroll_2$ , respectively

## Boolean set operations (union, intersection, and set difference)

- Let  $E_1$  and  $E_2$  be two RA expressions with the **same** schema then
  - $E_1 \cup E_2$ ,
  - $E_1 \cap E_2$ , and
  - $E_1 - E_2$
- are RA expressions with the same schema
- If  $Q_{E_1}$  and  $Q_{E_2}$  are the SQL queries corresponding to  $E_1$  and  $E_2$ , respectively, then the expressions are expressed in SQL by
  - $Q_{E_1} \text{ UNION } Q_{E_2}$ ,
  - $Q_{E_1} \text{ INTERSECT } Q_{E_2}$ , and
  - $Q_{E_1} \text{ EXCEPT } Q_{E_2}$

## Examples

- In this example, assume relation  $Enroll(sid, cno, grade)$
- "Find the sid of each student who received an 'A' in some course."

$$\pi_{sid}(\sigma_{grade='A'}(Enroll))$$

- "Find the sid of each student who received an 'A' in some course and a 'B' in some course."

$$\pi_{sid}(\sigma_{grade='A'}(Enroll)) \cap \pi_{sid}(\sigma_{grade='B'}(Enroll))$$

- "Find the sid of each student who did not receive an 'A' in any course."

$$\pi_{sid}(Student) - \pi_{sid}(\sigma_{grade='A'}(Enroll))$$

- "Find the sids of students who are enrolled in at least one course"

$$\pi_{sid}(Enroll)$$

- "Find the sids of students who take at least two courses"

$$\pi_{E1.sid}(\sigma_{E1.sid = E2.sid \wedge E1.cno \neq E2.cno}(Enroll_1 \times Enroll_2))$$

- "Find the sids of students who take exactly one course"

$$\pi_{sid}(Enroll) - \pi_{E1.sid}(\sigma_{E1.sid = E2.sid \wedge E.cno \neq E2.cno}(Enroll_1 \times Enroll_2))$$

## Examples (**NOT ONLY** and **ONLY**)

- "Find the cnos of CS courses."

$$\pi_{cno}(\sigma_{dept='CS}(Course))$$

Denote this expression by *CS*

- "Find sids of students who do **not only** take CS courses."

$$\pi_{sid}(Enroll - (\pi_{sid}(Student) \times CS))$$

- "Find sids of students who **only** take CS courses."

$$\pi_{sid}(Student) - \pi_{sid}(Enroll - (\pi_{sid}(Student) \times CS))$$

## Examples (**NOT ALL** and **ALL**)

- "Find sids of students who do **not** take **all** CS courses"

$$\pi_{sid}((\pi_{sid}(Student) \times CS) - Enroll)$$

- "Find sids of students who take **all** CS courses"

$$\pi_{sid}(Student) - \pi_{sid}(\pi_{sid}(Student) \times CS) - Enroll)$$

## Examples

Consider the relation  $\text{Person}(pid, age)$

- "Find the pids of persons who are not the youngest."

$$\pi_{P1.pid}(\sigma_{P1.age > P2.age}(\text{Person}_1 \times \text{Person}_2))$$

- "Find the pids of the youngest persons."

$$\pi_{pid}(\text{Person}) - \pi_{P1.pid}(\sigma_{P1.age > P2.age}(\text{Person}_1 \times \text{Person}_2))$$

# Translating SQL Queries to Relational Algebra Expressions

Dirk Van Gucht<sup>1</sup>

<sup>1</sup>Indiana University

February 26, 2019

## Outline

- **Objective:** Discuss an algorithm that translates a SQL query into an equivalent RA expression
- **Motivation:** Translate a declaratively specified query into a procedurally specified query
- **Restriction:** We do not attempt to get an efficient RA expression. Finding an equivalent efficient RA expression is done during **query optimization**

## Strategy

- SQL queries with set predicates will be translated to equivalent SQL queries without set predicates
- WHERE conditions will be eliminated by translating them into FROM clauses using selections and join operations, or by decomposing them into more basic components (then translate these) and then use the set operations union, intersection, and set difference
- These SQL queries will then be translated into RA expressions

## Query Forms (Basic case)

```
SELECT [DISTINCT] L( $t_1, \dots, t_n$ )
FROM   R1  $t_1, \dots, R_n t_n$ 
WHERE  C( $t_1, \dots, t_n$ )
```

- $L(t_1, \dots, t_n)$  is a list of (named) components of the tuple variables  $t_1$  through  $t_n$ <sup>1</sup>
- $R_1$  through  $R_n$  are either relations or non-parameterized SQL queries aliased by the tuple variables  $t_1$  through  $t_n$
- $C(t_1, \dots, t_n)$  is any valid SQL condition involving the components of the variables  $t_1$  through  $t_n$
- We do not handle SQL queries with aggregate functions or queries with subqueries in the FROM clause

---

<sup>1</sup>There may also appear constants in  $L$ ; these need special treatment

## Query Forms (queries with set operations)

Assuming that  $Q$ ,  $Q_1$ , and  $Q_2$  are SQL queries, we consider the following SQL query forms

$$\begin{array}{c} Q_1 \\ \text{UNION} \\ Q_2 \end{array}$$
$$\begin{array}{c} Q_1 \\ \text{INTERSECT} \\ Q_2 \end{array}$$
$$\begin{array}{c} Q_1 \\ \text{EXCEPT} \\ Q_2 \end{array}$$
$$(Q)$$

## Discussion: interaction between projection $\pi$ and set operations $\cup$ , $\cap$ , and $-$

- Before we can start with the translation algorithm, it is crucial to discuss how the projection operator  $\pi$  interacts with the set operations  $\cup$ ,  $\cap$ , and  $-$ .
- Understanding this is vital for the correct translations of SQL **WHERE** clauses that use the **OR**, **AND**, and **NOT** boolean operations
- We will consider the following interactions:
  - Projection  $\pi$  and union  $\cup$
  - Projection  $\pi$  and intersection  $\cap$
  - Projection  $\pi$  and set difference  $-$

## Projection $\pi$ distributes over $\cup$

Given RA expressions  $E_1$  and  $E_2$ , it is the case that

$$\pi_L(E_1 \cup E_2) = \pi_L(E_1) \cup \pi_L(E_2)$$

An application of this is the following: ( $C_1$  and  $C_2$  are some conditions)

$$\pi_L(\sigma_{C_1 \vee C_2}(E)) = \pi_L(\sigma_{C_1}(E) \cup \sigma_{C_2}(E)) = \pi_L(\sigma_{C_1}(E)) \cup \pi_L(\sigma_{C_2}(E))$$

## Translating OR in WHERE clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C1(t1, ..., tn) OR C2(t1, ..., tn)
```

can be translated to become

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C1(t1, ..., tn)
UNION
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C2(t1, ..., tn)
```

## Projection $\pi$ does not distribute over $\cap$

Given RA expressions  $E_1$  and  $E_2$ , it is the case that

$$\pi_L(E_1 \cap E_2) \subseteq \pi_L(E_1) \cap \pi_L(E_2)$$

But there exist cases where

$$\pi_L(E_1 \cap E_2) \neq \pi_L(E_1) \cap \pi_L(E_2)$$

This complexity features in reasoning about the expression  $\pi_L(\sigma_{C_1 \wedge C_2}(E))$  since

$$\pi_L(\sigma_{C_1 \wedge C_2}(E)) = \pi_L(\sigma_{C_1}(E) \cap \sigma_{C_2}(E)) \subseteq \pi_L(\sigma_{C_1}(E)) \cap \pi_L(\sigma_{C_2}(E))$$

But there are cases where

$$\pi_L(\sigma_{C_1 \wedge C_2}(E)) \neq \pi_L(\sigma_{C_1}(E)) \cap \pi_L(\sigma_{C_2}(E))$$

## Projection $\pi$ does **not** distribute over intersection $\cap$

| R |   |   |
|---|---|---|
| a | b | c |
| 1 | 1 | 2 |
| 1 | 2 | 1 |

$$\pi_a(\sigma_{b=1 \wedge c=1}(R))$$

|   |
|---|
| a |
|   |

correct

$$\pi_a(\sigma_{b=1}(R) \cap \sigma_{c=1}(R))$$

|   |
|---|
| a |
|   |

incorrect

$$\pi_a(\sigma_{b=1}(R)) \cap \pi_a(\sigma_{c=1}(R))$$

|   |
|---|
| a |
| 1 |

## Projection $\pi$ does not distribute over intersection $\cap$ (Predicate Logic)

The underlying reason why projection  $\pi$  does not distribute over intersection  $\cap$  is that an existential quantifier  $\exists$  does not distribute over a conjunction  $\wedge$  in Predicate Logic

$$\begin{aligned}\pi_a(\sigma_{b=1} \wedge c=1(R)) \\ &= \\ \pi_a(\sigma_{b=1}(R) \cap \sigma_{c=1}(R)) \\ &= \\ \{a \mid \exists b \exists c [(R(a, b, c) \wedge b = 1) \wedge (R(a, b, c) \wedge c = 1)]\} \\ &\quad \textcolor{red}{\subset} \\ \{a \mid \exists b \exists c (R(a, b, c) \wedge b = 1) \wedge \exists b \exists c (R(a, b, c) \wedge c = 1)\} \\ &= \\ \pi_{b=1}(R) \cap \pi_{c=1}(R)\end{aligned}$$

## Translating AND in WHERE clause (Correct)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE C1(t1, ..., tn) AND C2(t1, ..., tn)
```

can be translated to become

```
SELECT Lq(t1, ..., tn)2
FROM (SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn
       WHERE C1(t1, ..., tn)
       INTERSECT
       SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn
       WHERE C2(t1, ..., tn)
) q
```

---

<sup>2</sup>  $L^q(t_1, \dots, t_n)$  indicates that the components of  $t_1$  through  $t_n$  in  $L$  may need to be renamed as components of  $q$

## Translating AND in WHERE clause (Incorrect)

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Rn tn
WHERE  C1(t1, ..., tn) AND C2(t1, ..., tn)
```

is not equivalent<sup>3</sup> with

```
SELECT      L(t1, ..., tn)
FROM        R1 t1, ..., Rn tn
WHERE       C1(t1, ..., tn)
INTERSECT
SELECT      L(t1, ..., tn)
FROM        R1 t1, ..., Rn tn
WHERE       C2(t1, ..., tn)
```

---

<sup>3</sup>Projection does not distribute over intersection

## Translating AND in WHERE clause (Correct)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND e.grade = 'A'
```

can be translated to become

```
SELECT q.sid  
FROM (SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.cno = 100  
INTERSECT  
      SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.grade = 'A'  
) q
```

## Translating AND in WHERE clause (Incorrect)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND e.grade = 'A'
```

is not equivalent<sup>4</sup> with

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100  
INTERSECT  
SELECT e.sid  
FROM Enroll e  
WHERE e.grade = 'A'
```

---

<sup>4</sup>Projection does not distribute over intersection

## Projection $\pi$ does not distribute over $-$

Given RA expressions  $E_1$  and  $E_2$ , it is the case that

$$\pi_L(E_1 - E_2) \supseteq \pi_L(E_1) - \pi_L(E_2)$$

But there exist cases where

$$\pi_L(E_1 - E_2) \neq \pi_L(E_1) - \pi_L(E_2)$$

This complexity features in reasoning about the expression  $\pi_L(\sigma_{\neg C}(E))$  since

$$\pi_L(\sigma_{\neg C}(E)) = \pi_L(E - \sigma_C(E)) \supseteq \pi_L(E) - \pi_L(\sigma_C(E))$$

But there are cases where

$$\pi_L(\sigma_{\neg C}(E)) \neq \pi_L(E) - \pi_L(\sigma_C(E))$$

## Projection $\pi$ does **not** distribute over set difference –

| R |   |
|---|---|
| a | b |
| 1 | 1 |
| 1 | 2 |

$$\pi_a(\sigma_{\neg(b=1)}(R))$$

|   |
|---|
| a |
| 1 |

correct

$$\pi_a(R - \sigma_{b=1}(R))$$

|   |
|---|
| a |
| 1 |

incorrect

$$\pi_a(R) - \pi_a(\sigma_{b=1}(R))$$

|   |
|---|
| a |
|   |

## Projection $\pi$ does not distribute over set difference – (Predicate Logic)

The underlying reason why projection  $\pi$  does not distribute over distribution  $\cap$  is that an existential quantifier  $\exists$  does not distribute over a conjunction-negation  $\wedge \neg$  sequence in Predicate Logic

$$\begin{aligned}\pi_a(\sigma_{\neg(b=1)}(R)) \\ &= \\ \pi_a(R - \sigma_{b=1}(R)) \\ &= \\ \{a \mid \exists b (R(a, b) \wedge \neg(R(a, b) \wedge b = 1))\} \\ &\supsetneq \\ \{a \mid \exists b R(a, b) \wedge \neg \exists b (R(a, b) \wedge b = 1)\} \\ &= \\ \pi_a(R) - \pi_a(\sigma_{b=1}(R))\end{aligned}$$

## Translating NOT in WHERE clause (Correct)

```
SELECT  L(t1,...,tn)
FROM    R1 t1,...,Rn tn
WHERE   NOT C(t1,...,tn)
```

can be translated to become

```
SELECT  Lq(t1,...,tn)
FROM    (SELECT  t1.*,...,tn.*
          FROM    R1 t1,...,Rn tn
          EXCEPT
          SELECT  t1.*,...,tn.*
          FROM    R1 t1,...,Rn tn
          WHERE   C(t1,...,tn)
) q
```

## Translating NOT in WHERE clause (Incorrect)

```
SELECT  L(t1, ..., tn)
FROM    R1 t1, ..., Rn tn
WHERE   NOT C(t1, ..., tn)
```

is not equivalent<sup>5</sup> with

```
SELECT  L(t1, ..., tn)
FROM    R1 t1, ..., Rn tn
EXCEPT
SELECT  L(t1, ..., tn)
FROM    R1 t1, ..., Rn tn
WHERE   C(t1, ..., tn)
```

---

<sup>5</sup>Projection do not distribute over set difference

## Translating NOT in WHERE clause (Correct)

```
SELECT e.sid  
FROM Enroll e  
WHERE NOT e.grade = 'A'
```

can be translated to become

```
SELECT q.sid  
FROM (SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      EXCEPT  
      SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.grade = 'A'  
    ) q
```

## Translating NOT in WHERE clause (Incorrect)

```
SELECT e.sid  
FROM Enroll e  
WHERE NOT e.grade = 'A'
```

is not equivalent<sup>6</sup> with

```
SELECT e.sid  
FROM Enroll e  
EXCEPT  
SELECT e.sid  
FROM Enroll e  
WHERE e.grade = 'A'
```

---

<sup>6</sup>Projection does not distribute over set difference

## Projection $\pi$ does not distribute over set difference –

| R |   |   |
|---|---|---|
| A | B | C |
| 1 | 1 | 2 |
| 1 | 2 | 1 |

$$\pi_A(\sigma_{B=1 \wedge \neg(C=1)}(R))$$

|   |
|---|
| A |
| 1 |

correct

$$\pi_A(\sigma_{B=1}(R) - \sigma_{C=1}(R))$$

|   |
|---|
| A |
| 1 |

incorrect

$$\pi_A(\sigma_{B=1}(R)) - \pi_A(\sigma_{C=1}(R))$$

|   |
|---|
| A |
|   |

## Translating AND NOT in WHERE clause (Correct)

```
SELECT  L( $t_1, \dots, t_n$ )
FROM     $R_1 t_1, \dots, R_n t_n$ 
WHERE    $C_1(t_1, \dots, t_n)$  AND NOT  $C_2(t_1, \dots, t_n)$ 
```

can be translated to become

```
SELECT   $L^q(t_1, \dots, t_n)$ 
FROM    (SELECT       $t_1.*, \dots, t_n.*$ 
          FROM         $R_1 t_1, \dots, R_n t_n$ 
          WHERE        $C_1(t_1, \dots, t_n)$ 
          EXCEPT
          SELECT       $t_1.*, \dots, t_n.*$ 
          FROM         $R_1 t_1, \dots, R_n t_n$ 
          WHERE        $C_2(t_1, \dots, t_n)$ 
        ) q
```

## Translating AND NOT in WHERE clause (Incorrect)

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Rn tn
WHERE  C1(t1, ..., tn) AND NOT C2(t1, ..., tn)
```

is not equivalent<sup>7</sup> with

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Rn tn
WHERE  C1(t1, ..., tn)
EXCEPT
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Rn tn
WHERE  C2(t1, ..., tn)
```

---

<sup>7</sup>Projection do not distribute over set difference

## Translating AND NOT in WHERE clause (Correct)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND NOT e.grade = 'A'
```

can be translated to become

```
SELECT q.sid  
FROM (SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.cno = 100  
      EXCEPT  
      SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.grade = 'A'  
) q
```

## Translating AND NOT in WHERE clause (Incorrect)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND NOT e.grade = 'A'
```

is not equivalent<sup>8</sup> with

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100  
EXCEPT  
SELECT e.sid  
FROM Enroll e  
WHERE e.grade = 'A'
```

---

<sup>8</sup>Projection does not distribute over set difference

## Translating AND NOT in WHERE clause (Also correct)

```
SELECT  L( $t_1, \dots, t_n$ )
FROM     $R_1 t_1, \dots, R_n t_n$ 
WHERE    $C_1(t_1, \dots, t_n)$  AND NOT  $C_2(t_1, \dots, t_n)$ 
```

can be also be translated to become

```
SELECT  Lq( $t_1, \dots, t_n$ )
FROM    (SELECT       $t_1.*, \dots, t_n.*$ 
        FROM        $R_1 t_1, \dots, R_n t_n$ 
        WHERE      $C_1(t_1, \dots, t_n)$ 
        EXCEPT
        SELECT       $t_1.*, \dots, t_n.*$ 
        FROM        $R_1 t_1, \dots, R_n t_n$ 
        WHERE      $C_1(t_1, \dots, t_n)$  AND  $C_2(t_1, \dots, t_n)$ 
) q
```

## Translating AND NOT in WHERE clause (Also correct)

```
SELECT e.sid  
FROM Enroll e  
WHERE e.cno = 100 AND NOT e.grade = 'A'
```

can be translated to become

```
SELECT q.sid  
FROM (SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.cno = 100  
      EXCEPT  
      SELECT e.sid, e.cno, e.grade  
      FROM Enroll e  
      WHERE e.cno= 100 AND e.grade = 'A'  
      ) q
```

## Translating EXISTS in WHERE clause (Example)

Let  $R(a, b)$  and  $S(b, c)$  be two relations.

```
SELECT r.a
FROM R r
WHERE EXISTS (SELECT S.c
               FROM S s
               WHERE r.b = s.b)
```

In Predicate Logic,

$$\begin{aligned} & \{a \mid \exists b (R(a, b) \wedge \{c \mid S(b, c)\} \neq \emptyset)\} \\ &= \\ & \{a \mid \exists b (R(a, b) \wedge \exists c S(b, c))\} \\ &= \\ & \{a \mid \exists b \exists c (R(a, b) \wedge S(b, c))\} \\ &= \\ & \pi_{R.a}(\sigma_{R.b=S.b}(R \times S)) \end{aligned}$$

## Translating EXISTS in WHERE clause (Example in Predicate Logic)

$$\begin{aligned} & \{a \mid \exists b (R(a, b) \wedge \{c \mid S(b, c)\} \neq \emptyset)\} \\ &= \\ & \{a \mid \exists b (R(a, b) \wedge \exists c S(b, c))\} \\ &= \\ & \{a \mid \exists b \exists c (R(a, b) \wedge S(b, c))\} \\ &= \\ & \pi_{R.a}(\sigma_{R.b=S.b}(R \times S)) \end{aligned}$$

In SQL,

```
SELECT DISTINCT r.a
FROM R r, S s
WHERE r.b = s.b
```

## Translating EXISTS in WHERE clause (Example)

Let  $R(a, c)$  and  $S(b, c)$  be two relations.

```
SELECT r.a
FROM R r
WHERE EXISTS (SELECT S.c
               FROM S s
               WHERE r.b = s.b)
```

is translated to

```
SELECT DISTINCT r.a
FROM R r, S s
WHERE r.b = s.b
```

## Translating EXISTS in WHERE clause (General case)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE EXISTS (SELECT 1
               FROM S1 u1, ..., Sm um
               WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT DISTINCT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
WHERE C(u1, ..., um, t1, ..., tn)
```

## Translating EXISTS in WHERE clause

```
SELECT s.sid
FROM Student s
WHERE EXISTS (SELECT 1
               FROM Enroll e, Course c
               WHERE e.sid = s.sid AND e.cno = c.cno AND c.dept = 'CS')
```

is translated to

```
SELECT DISTINCT s.sid
FROM Student s, Enroll e, Course c
WHERE e.sid = s.sid AND e.cno = c.cno AND c.dept = 'CS'
```

## Translating NOT EXISTS in WHERE clause (Example)

Let  $R(a, b)$  and  $S(b, c)$  be two relations.

```
SELECT r.a
FROM R r
WHERE NOT EXISTS (SELECT S.c
                    FROM S s
                    WHERE r.b = s.b)
```

In Predicate Logic,

$$\begin{aligned} & \{a \mid \exists b (R(a, b) \wedge \{c \mid S(b, c)\} = \emptyset)\} \\ &= \\ & \{a \mid \exists b (R(a, b) \wedge \neg \exists c S(b, c)))\} \\ &= \\ & \{a \mid \exists b (R(a, b) \wedge \neg \exists c (R(a, b) \wedge S(b, c)))\} \\ &= \\ & \pi_{R.a}(R - \pi_{R.a, R.b}(\sigma_{R.b=S.b}(R \times S))) \end{aligned}$$

## Translating NOT EXISTS in WHERE clause (Example)

$$\begin{aligned} & \{a \mid \exists b(R(a, b) \wedge \{c \mid S(b, c)\} = \emptyset)\} \\ &= \\ & \pi_{R.a}(R - \pi_{R.a, R.b}(\sigma_{R.b=S.b}(R \times S))) \end{aligned}$$

In SQL,

```
SELECT DISTINCT q.a
FROM   (SELECT      R.a, R.b
        FROM       R r
        EXCEPT
        SELECT      R.a, R.b
        FROM       R r, S s
        WHERE      r.b = s.b
      ) q
```

## Translating NOT EXISTS in WHERE clause (Example)

```
SELECT r.a
FROM R r
WHERE NOT EXISTS (SELECT S.c
                   FROM S s
                   WHERE r.b = s.b)
```

is translated to

```
SELECT DISTINCT q.a
FROM (SELECT      R.a, R.b
      FROM       R r
      EXCEPT
      SELECT      R.a, S.b
      FROM       R r, S s
      WHERE      r.b = s.b
) q
```

## Translating NOT EXISTS in WHERE clause (General case)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE NOT EXISTS (SELECT 1
                   FROM S1 u1, ..., Sm um
                   WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT DISTINCT Lq(t1, ..., tn)
FROM (SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn
       EXCEPT
       SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn)
     ) q
```

## Translating NOT EXISTS in WHERE clause

```
SELECT s.sid  
FROM Student s  
WHERE NOT EXISTS (SELECT 1  
                   FROM Enroll e  
                   WHERE e.sid = s.sid AND e.grade = 'A')
```

is translated to

```
SELECT q.sid  
FROM (SELECT s.sid, s.sname  
      FROM Student s  
      EXCEPT  
      SELECT s.sid, s.sname  
      FROM Student s, Enroll e  
      WHERE e.sid = s.sid AND e.grade = 'A'  
) q
```

## Translating AND NOT EXISTS in WHERE clause

```
SELECT s.sid
FROM Student s
WHERE s.sname = 'Ann' AND
      NOT EXISTS (SELECT 1
                  FROM Enroll e
                  WHERE e.sid = s.sid AND e.grade = 'A')
```

is translated to

```
SELECT q.sid
FROM (SELECT s.sid, s.sname
      FROM Student s
      WHERE s.sname = 'Ann'
      EXCEPT
      SELECT s.sid, s.sname
      FROM Student s, Enroll e
      WHERE e.sid = s.sid AND e.grade = 'A'
) q
```

## Translating AND NOT EXISTS in WHERE clause (Alternative)

```
SELECT s.sid
FROM Student s
WHERE s.sname = 'Ann' AND
NOT EXISTS (SELECT 1
            FROM Enroll e
            WHERE e.sid = s.sid AND e.grade = 'A')
```

is translated to

```
SELECT q.sid
FROM (SELECT s.sid, s.sname
      FROM Student s
      WHERE s.sname = 'Ann'
EXCEPT
      SELECT s.sid, s.sname
      FROM Student s, Enroll e
      WHERE s.sname='Ann' AND e.sid = s.sid AND e.grade = 'A'
) q
```

## Translating IN in WHERE clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE (ti1.Aj1, ..., tik.Ajk) IN
      (SELECT ul1.Bm1, ..., ulk.Bmk
       FROM S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT DISTINCT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
WHERE C(u1, ..., um, t1, ..., tn) AND
      ti1.Aj1 = ul1.Bm1 AND ... AND tik.Ajk = ulk.Bmk
```

## Translating NOT IN in WHERE clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE (ti1.Aj1, ..., tik.Ajk) NOT IN
      (SELECT ul1.Bm1, ..., ulk.Bmk
       FROM S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT Lq(t1, ..., tn)
FROM (SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn
       EXCEPT
       SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn) AND
             ti1.Aj1 = ul1.Bm1 AND ... AND tik.Ajk = ulk.Bmk
) q
```

## Translating $\theta$ SOME in WHERE clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE ti1.Aj1  $\theta$  SOME
      (SELECT ul1.Bm1
       FROM S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT DISTINCT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
WHERE C(u1, ..., um, t1, ..., tn) AND
      ti1.Aj1  $\theta$  ul1.Bm1
```

## Translating $\theta$ ALL in WHERE clause

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Rn tn
WHERE ti1.Aj1  $\theta$  ALL
      (SELECT ui1.Bm1
       FROM S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn))
```

is translated to

```
SELECT Lq(t1, ..., tn)
FROM (SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn
       EXCEPT
       SELECT t1.* , ..., tn.*
       FROM R1 t1, ..., Rn tn, S1 u1, ..., Sm um
       WHERE C(u1, ..., um, t1, ..., tn) AND
             NOT ti1.Aj1  $\theta$  ui1.Bm1
      ) q
```

## Translating $\theta$ ALL in WHERE clause

```
SELECT p.pid
FROM Person p
WHERE p.age ≤ ALL
      (SELECT p1.age
       FROM Person p1)
```

is translated to

```
SELECT DISTINCT q.pid
FROM (SELECT p.pid, p.age
      FROM Person p
EXCEPT
      SELECT p.pid, p.age
      FROM Person p, Person p1
      WHERE NOT p.age ≤ p1.age
) q
```

## Translating SQL with doubly-nested set predicates

“Find the sid of each student who is only enrolled CS courses.”

**Eliminate first-level NOT EXISTS predicate**

“Find the sid of each student who is only enrolled in CS courses.”

## Eliminate second-level AND NOT EXISTS

“Find the sid of each student who is only enrolled in CS courses.”

```
SELECT      q1.sid
FROM        (SELECT s.sid AS sid, s.sname
            FROM  Student s
            EXCEPT
            SELECT q2.sid, s.sname
            FROM (SELECT s.sid AS sid, s.sname, e.sid, e.cno, e.grade
                  FROM  Student s, Enroll e
                  WHERE e.sid = s.sid
                  EXCEPT
                  (SELECT s.sid, s.sname, e.sid, e.cno, e.grade
                  FROM  Student s, Enroll e, Course c
                  WHERE e.cno = c.cno AND c.dept = 'CS' ) q2) q1
```

## Translating SQL with doubly-nested set predicates

“Find the sid of each student who is enrolled in all CS courses.”



## Eliminate second-level AND NOT EXISTS

"Find the sid of each student who is enrolled in all CS courses."

```
SELECT q1.ssid
FROM (SELECT s.sid AS ssid, s.sname
      FROM Student s
      EXCEPT
      SELECT q2.ssid, s.sname
      FROM (SELECT s.sid AS ssid, s.sname, c.cno, c.dname
            FROM Student s, Course c
            WHERE c.dname = 'CS'
            EXCEPT
            (SELECT s.sid, s.sname, c.cno, c.dname
             FROM Student s, Enroll e, Course c
             WHERE e.cno = c.cno AND e.sid = s.sid ) q2) q1
```

## Moving **WHERE** condition to **FROM** clause

- In the previous slides we have shown how set predicates can be translated
- After that process, we will have SQL queries wherein the **WHERE** clauses consist of boolean combinations of conditions of the form
  - $t.A \theta a$ ; or
  - $t_i.A \theta t_j.B$
- In the following slides we will show how these **WHERE** clauses can be moved to **FROM** clauses
- We might also have queries without a **WHERE** clause and/or without a **FROM** clause; these require special treatment

## SQL queries without **WHERE** and **FROM** clauses

SELECT a AS A

This query is translated to the RA expression

$(A : a)$

## SQL queries without **FROM** clauses

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Rn tn
```

This query is translated to the RA expression

$$\pi_{L(t_1, \dots, t_n)}(R_1 \times \dots \times R_n)$$

## Moving WHERE condition to FROM clause (condition on at least three relations)

Assume that the condition  $C$  applies to at least three relations, i.e., the condition is of the form  $C(t_{i_1}, t_{i_2}, t_{i_3}, \dots, t_{i_k})$  with  $k \geq 3$

```
SELECT L(t1, ..., tn)
FROM R1 t1, R2 t2, R3 t3, ..., Rn tn
WHERE C(ti1, ti2, ti3, ..., tik)
```

We can now introduce the CROSS JOIN in the FROM clause by replacing each ‘,’ with CROSS JOIN

```
SELECT L(t1, ..., tn)
FROM R1 t1 CROSS JOIN R2 t2 CROSS JOIN R3 t3 CROSS JOIN ... CROSS JOIN Rn tn
WHERE C(ti1, ti2, ti3, ..., tik)
```

In RA,

$$\pi_{L(t_1, \dots, t_n)}(\sigma_{C(t_{i_1}, t_{i_2}, t_{i_3}, \dots, t_{i_k})}(R_1 \times R_2 \times R_3 \times \dots \times R_n))$$

## Moving WHERE condition to FROM clause (condition on at least three relations) Example

```
SELECT L(t1, t2, t3)
FROM R1 t1, R2 t2, R3 t3
WHERE t1.A1 θ1 t2.A2 OR t2.A3 θ2 t3.A4
```

is translated to

$$\pi_{L(t_1, t_2, t_3)}(\sigma_{t_1.A_1 \theta_1 t_2.A_2 \vee t_2.A_3 \theta_2 t_3.A_4}(R_1 \times R_2 \times R_3))$$

## Moving WHERE condition to FROM clause (condition on single relation)

```
SELECT L(t1, ..., tn)
FROM R1 t1, R2 t2, ..., Ri ti, ..., Rn tn
WHERE C(ti) [AND C'(ti1, ..., tik)]
```

Observe that  $C(t_i)$  is only a condition on  $R_i$ . This query is translated to

```
SELECT L(t1, ..., tn)
FROM R1 t1, R2 t2 ...,
      (SELECT ti.* FROM Ri WHERE C(ti)) ti, ..., Rn tn
[WHERE C'(ti1, ..., tik)]
```

## Moving WHERE condition to FROM clause (condition on single relation)

```
SELECT L(t1, ..., tn)
FROM   R1 t1, R2 t2, ...,
       (SELECT ti.* FROM Ri WHERE C(ti)) ti, ..., Rn tn
[WHERE  C'(ti1, ..., tik)]
```

We can now introduce the CROSS JOIN in the FROM clause by replacing each ‘,’ with CROSS JOIN

```
SELECT L(t1, ..., tn)
FROM   R1 t1 CROSS JOIN R2 t2 CROSS JOIN ... CROSS JOIN
       (SELECT ti.* FROM Ri WHERE C(ti)) ti CROSS JOIN ... CROSS JOIN Rn tn
[WHERE  C'(ti1, ..., tik)]
```

Which, in the notation of RA, corresponds to the expression

$$\pi_{L(t_1, \dots, t_n)}(\sigma_{C'(t_{i_1}, \dots, t_{i_k})}(R_1 \times R_2 \times \dots \times \sigma_{C(t_i)}(R_i) \times \dots \times R_n))$$

or, when  $C'(t_{i_1}, \dots, t_{i_k})$  is missing,

$$\pi_{L(t_1, \dots, t_n)}(R_1 \times R_2 \times \dots \times \sigma_{C(t_i)}(R_i) \times \dots \times R_n)$$

## Moving WHERE condition to FROM clause (condition on two relations)

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Ri ti, ..., Rj tj, ..., Rn tn
WHERE C(ti, tj) [AND C'(ti1, ..., tik)]
```

Observe that  $C(t_i, t_j)$  is a condition relating  $R_i$  and  $R_j$ . This query is translated to

```
SELECT L(t1, ..., tn)
FROM R1 t1, ..., Ri-1 tj-1, Ri+1 ti+1, ..., Rj-1 tj-1, Rj+1 tj+1, ..., Rn tn,
      Ri ti JOIN Rj tj ON C(ti, tj)
[WHERE C'(ti1, ..., tik)]
```

## Moving WHERE condition to FROM clause (condition on two relations)

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Ri-1 tj-1, Ri+1 ti+1, ..., Rj-1 tj-1, Rj+1 tj+1, ... Rn tn,
       Ri ti JOIN Rj tj ON C(ti, tj)
[WHERE C'(ti1, ..., tik)]
```

Recalling that each ‘,’ in the FROM clause corresponds to a CROSS JOIN, this query can be formulated in RA as follows:

$$\pi_{L(t_1, \dots, t_n)}(\sigma_{C'(t_{i_1}, \dots, t_{i_k})}(R_1 \times \dots \times R_{i-1} \times R_{i+1} \times \dots \times R_{j-1} \times R_{j+1} \times \dots \times R_n \times (R_i \bowtie_{C(t_i, t_j)} R_j)))$$

or, when  $C'(t_{i_1}, \dots, t_{i_k})$  is missing

$$\pi_{L(t_1, \dots, t_n)}(R_1 \times \dots \times R_{i-1} \times R_{i+1} \times \dots \times R_{j-1} \times R_{j+1} \times \dots \times R_n \times (R_i \bowtie_{C(t_i, t_j)} R_j))$$

## Moving WHERE condition to FROM clause (natural join condition on two relations)

Assume that  $A_1, \dots, A_k$  are the common attributes of  $R_i$  and  $R_j$  and that  $C(t_i, t_j)$  is the condition

$$t_i.A_1 = t_j.A_1 \text{ AND } \dots \text{ AND } t_i.A_k = t_j.A_k$$

then

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Ri-1 tj-1, Ri+1 ti+1, ..., Rj-1 tj-1, Rj+1 tj+1, ..., Rn tn,
       Rj ti JOIN Rj tj ON C(ti, tj)
[WHERE C'(ti1, ..., tik)]
```

is translated to

```
SELECT L(t1, ..., tn)
FROM   R1 t1, ..., Ri-1 tj-1, Ri+1 ti+1, ..., Rj-1 tj-1, Rj+1 tj+1, ..., Rn tn,
       Rj ti NATURAL JOIN Rj tj
[WHERE C'(ti1, ..., tik)]
```

In RA,

$$\pi_{L(t_1, \dots, t_n)}(\sigma_{C'(t_{i_1}, \dots, t_{i_k})}(R_1 \times \dots \times R_{i-1} \times R_{i+1} \times \dots \times R_{j-1} \times R_{j+1} \times \dots \times R_n \times (R_i \bowtie R_j)))$$

## SQL queries with set operations **UNION**, **INTERSECT**, or **EXCEPT**

Assuming  $Q_1$  and  $Q_2$  SQL queries, the queries of the form

$$\begin{matrix} Q_1 \\ \text{UNION [INTERSECT | EXCEPT]} \\ Q_2 \end{matrix}$$

can be translated to RA as follows

$$E_{Q_1} \cup [\cap | -] E_{Q_2}$$

where  $E_{Q_1}$  and  $E_{Q_2}$  are the RA expressions corresponding to  $Q_1$  and  $Q_2$

## Example

“Find the sid of each student who is only enrolled in CS courses.”

```
SELECT    q1.sid
FROM      (SELECT s.sid AS sid, s.sname
           FROM Student s
         EXCEPT
           SELECT q2.sid, s.sname
             FROM (SELECT s.sid AS sid, s.sname, e.sid, e.cno
                   FROM Student s, Enroll e
                  WHERE e.sid = s.sid
                EXCEPT
                  (SELECT s.sid, s.sname, e.sid, e.cno
                    FROM Student s, Enroll e, Course c
                   WHERE e.cno = c.cno AND c.dept = 'CS' ) q2) q1
```

## Example

“Find the sid of each student who is only enrolled in CS courses.”

```
SELECT      q1.sid
FROM        (SELECT s.sid AS ssid, s.sname
            FROM Student s
           EXCEPT
          SELECT q2.sid, s.sname
            FROM (SELECT s.sid AS ssid, s.sname, e.sid, e.cno
                  FROM Student s NATURAL JOIN Enroll e
                 EXCEPT
                  (SELECT s.sid, s.sname, e.sid, e.cno
                    FROM Student s CROSS JOIN Enroll E
                     NATURAL JOIN (SELECT c.* FROM Course c WHERE dept = 'CS') c) q2) q1
```

is translated to

$$\pi_{sid}(\pi_{sid, sname}(S) - \pi_{S.sid, sname}(\pi_{S.sid, sname, E.sid, cno}(S \bowtie E) - \pi_{S.sid, sname, E.sid, cno}(S \times E \bowtie \sigma_{dept='CS'}(C))))$$

where  $S$ ,  $E$ , and  $C$  denote Student, Enroll, and Course, respectively.

## Example (Optimization)

“Find the sid of each student who is only enrolled in CS courses.”

$$\pi_{sid}(\pi_{sid, sname}(S) - \pi_{S.sid, sname}(\pi_{S.sid, sname, E.sid, cno}(S \bowtie E) - \pi_{S.sid, sname, E.sid, cno}(S \times E \bowtie \sigma_{dept='CS'}(C))))$$

This can be **optimized** to the RA expression

$$\pi_{sid}(S) - \pi_{sid}(\pi_{sid, cno}(E) - \pi_{sid, cno}(E) \ltimes \pi_{cno}(\sigma_{dept='CS'}(C)))$$

Notice that this is the RA expression for the **only** set semijoin  
If furthermore the schema of Enroll is (sid,cno), this expression becomes

$$\pi_{sid}(S) - \pi_{sid}(E - E \ltimes CS) = \pi_{sid} - \pi_{sid}(E \overline{\ltimes} CS)$$

where  $CS$  denotes the RA expression  $\pi_{cno}(\sigma_{dept='CS'}(C))$

# Joins and semijoins in relational algebra and SQL

Dirk Van Gucht<sup>1</sup>

<sup>1</sup>Indiana University

## Joins (Discovering Relationships)

- One of the main purposes of query languages, and more generally search languages, is to **discover relationships between objects in the data(base)**.
- Operations in query languages that can aid in discovering such relationships are called **joins**

## Joins (Discovering Relationships)

- In our running example, the database has three kinds of objects, i.e., **students**, **courses**, and **teachers**, and two kinds of relationships, i.e., **Enroll** relationships between students and courses, and **TaughtBy** relationships between courses and teachers.
- Starting from this data, we may then wish to discover each relationship  $(s, t)$  between a student  $s$  and a teacher  $t$  indicating that
  - student  $s$  takes **some** course taught by teacher  $t$ , or
  - student  $s$  takes **all** courses taught by teacher  $t$ , or
  - student  $s$  takes **two** courses taught by teacher  $t$ ,
  - etc.

## Semijoins (Discovering objects satisfying properties on the basis of relationships)

- Another main purpose of query languages, and more generally search languages, is to **discover objects in the database that satisfy certain properties**, in particular those that can only be established on the basis of relationships of these objects with other objects in the database.

For example

- Find each student  $s$  who is enrolled in a class taught by teacher 'Eric'
  - Find each teacher  $t$  who teaches no students who take CS courses
- 
- Operations in query languages that can aid in discovering objects with such properties are called **semijoins**

## Joins

- Two types of joins
  - Regular joins:<sup>1</sup> these are joins that compute relationships between objects ( $o_1, o_2$ ) based on tuple-component comparisons, like  $t_1.A \theta t_2.B$ , where  $o_1$  are components of  $t_1$  and  $o_2$  are components of  $t_2$ .
  - Set joins: these are joins that compute relations between objects based on comparisons between sets of tuples associated with these objects, e.g, a comparison between the set of courses taken by a student and the set of courses taught by a teacher.
- Analogously, there are also two types of semijoins.

---

<sup>1</sup>Regular joins are typically just called joins.

## Joins in SQL

- Regular joins: SQL supports regular joins with the special operators

|              |                              |
|--------------|------------------------------|
| JOIN         | or, equivalently, INNER JOIN |
| NATURAL JOIN | special case of JOIN         |
| CROSS JOIN   | cartesian product            |
|              | special case of NATURAL JOIN |

- Set joins: SQL has only limited support for set joins, but they can be simulated using the predicates

[NOT] EXISTS(… ) and [NOT] IN  
(SELECT COUNT(1) FROM … )  $\theta k$

## Semijoins in SQL

- Regular semijoins: SQL does not have special operations for semijoins, but they can be simulated with the **NATURAL JOIN** operations, or with the **IN** set predicate
- Set semijoins: SQL does not have set semijoins, but they can be shown to be special cases of set joins and as such can be simulated

## Regular joins in Relational Algebra (Introduction)

- Consider the query "Find the sid and age of each student who takes a course and whose name is Ann."
- This query can be expressed with the RA expression

$$\pi_{\text{Student.sid}, \text{age}}(\sigma_{\text{sname}=\text{'Ann'}} \wedge \text{Student.sid} = \text{Enroll.sid})(\text{Student} \times \text{Enroll})$$

- Or, using the notation  $S.sid$  instead of  $\text{Student}.sid$  and  $E.sid$  instead of  $\text{Enroll}.sid$ , more succinctly

$$\pi_{S.sid, \text{age}}(\sigma_{\text{sname}=\text{'Ann'}} \wedge S.sid = E.sid)(\text{Student} \times \text{Enroll})$$

## Regular joins in Relational Algebra (Introduction)

- Consider the query "Find the sid and age of each student who takes a course and whose name is Ann."
- This query can be expressed with the RA expression

$$\pi_{S.sid, \text{age}}(\sigma_{\text{pname}=\text{'Ann'}} \wedge S.sid=E.sid (\text{Student} \times \text{Enroll}))$$

- The subexpression

$$\sigma_{\text{pname}=\text{'Ann'}} \wedge S.sid=E.sid (\text{Student} \times \text{Enroll})$$

is called a (regular) join and we use the bowtie notation ( $\bowtie$ ) to write it as follows:

$$\text{Student} \bowtie_{\text{pname}=\text{'Ann'}} \wedge S.sid=E.sid \text{ Enroll}$$

- With the join operation, we can express the query as follow:

$$\pi_{S.sid, \text{age}}(\text{Student} \bowtie_{\text{pname}=\text{'Ann'}} \wedge S.sid=E.sid \text{ Enroll})$$

## Regular join in SQL (Introduction)

- "Find the sid and age of each student who takes a course and whose name is Ann."
- In SQL we can formulate this query as follows:

```
SELECT DISTINCT s.sid, s.age  
FROM Student s JOIN Enroll e  
ON (s.sname = 'Ann' AND s.sid = t.sid)
```

- As compared to the alternative SQL specification

```
SELECT DISTINCT s.sid, s.age  
FROM Student s, Enroll e  
WHERE s.sname = 'Ann' AND s.sid = t.sid
```

## Regular joins in the Relational Algebra (General case)

- Let  $E_1(A_1, \dots, A_m)$  and  $E_2(B_1, \dots, B_n)$  be RA expressions and let  $C$  be a condition involving the attributes  $A_1, \dots, A_m$  and  $B_1, \dots, B_n$ .
- Then

$$E_1 \bowtie_C E_2$$

is called the **join between  $E_1$  and  $E_2$  on condition  $C$** .

- The schema of this join is

$$(E_1.A_1, \dots, E_1.A_m, E_2.B_1, \dots, E_2.B_n)$$

- $E_1 \bowtie_C E_2$  is defined to be equivalent with the RA expression

$$\sigma_C(E_1 \times E_2)$$

## Regular joins in SQL (General case)

- If  $Q_{E_1}$  and  $Q_{E_2}$  denote the SQL queries corresponding to  $E_1$  and  $E_2$  then  $E_1 \bowtie_C E_2$  corresponds to the SQL query

```
SELECT  e1.A1, ..., e1.Am, e2.B1, ..., e2.Bn
FROM    (QE1) e1 JOIN (QE2) e2 ON (C)
```

- If there is no overlap between the schemas of  $Q_{E_1}$  and  $Q_{E_2}$ , the query can also be formulated as follows:

```
SELECT  A1, ..., Am, B1, ..., Bn
FROM    (QE1) JOIN (QE2) ON (C)
```

## More examples

- "Find the sids of students who take at least two courses."
- In RA

$$\pi_{E.sid}(Enroll \bowtie_{E.sid=E_1.sid \wedge E.cno \neq E_1.cno} Enroll_1)$$

- In SQL,

```
SELECT DISTINCT e.sid
FROM Enroll e JOIN Enroll e1
    ON (e.sid = e1.sid  $\wedge$  e.cno  $\neq$  e1.cno)
```

## More examples

- "Find the pids of persons who are not the youngest."
- In RA,

$$\pi_{P.pid}(Person \bowtie_{P.age < P_1.age} Person_1)$$

- In SQL,

```
SELECT DISTINCT p.pid
FROM Person p JOIN Person p1 ON (p.age < p1.age)
```

## Special case: cartesian product $\times$

- Let  $E_1(A_1, \dots, A_m)$  and  $E_2(B_1, \dots, B_n)$  be RA expressions
- Then,

$$E_1 \bowtie_{\text{true}} E_2 = E_1 \times E_2$$

- In SQL,

```
SELECT A1, ..., Am, B1, ..., Bn
FROM (QE1) JOIN (QE2) ON (true)
```

- Alternatively, in SQL, with CROSS JOIN

```
SELECT A1, ..., Am, B1, ..., Bn
FROM (QE1) CROSS JOIN (QE2)
```

## Special case: Natural joins (Introduction)

- A natural join between two relations is a series of "equality" joins on the common attributes of the relations, followed by a projection to remove redundant columns.
- Consider the relations  $\text{Enroll}(\text{sid}, \text{cno}, \text{grade})$  and  $\text{TaughtBy}(\text{cno}, \text{tid})$ ; notice that  $\text{cno}$  is a common attribute
- To perform the natural join between these relations, we write  $\text{Enroll} \bowtie \text{TaughtBy}$ , where

$\text{Enroll} \bowtie \text{TaughtBy} =$

$\pi_{E.\text{sid}, E.\text{cno}, E.\text{grade}, T.\text{tid}}(\text{Enroll} \bowtie_{E.\text{cno} = T.\text{cno}} \text{TaughtBy})$

- In SQL,

```
SELECT sid, cno, grade, tid  
FROM   Enroll NATURAL JOIN TaughtBy
```

## Multiple Natural joins

- Recall relations  $Enroll(sid, cno, grade)$ ,  
 $Student(sid, sname, age)$ , and  $Course(cno, cname, dept)$ .
- "Find the sid and sname of each student who takes a CS course."

$$\pi_{sid, sname}(Student \bowtie Enroll \bowtie \sigma_{dept='CS'}(Course))$$

- In SQL, this can be formulated as follows

```
WITH      CS_Course AS  
        (SELECT * FROM Course WHERE dept = 'CS')  
SELECT    DISTINCT sid, sname  
FROM     Student NATURAL JOIN Enroll  
          NATURAL JOIN CS_Course
```

## Natural join (General case)

- Let  $E_1(A_1, \dots, A_m, C_1, \dots, C_k)$  and  $E_2(C_1, \dots, C_k, B_1, \dots, B_n)$  be RA expressions
- Then

$$E_1 \bowtie E_2$$

is a **natural join** expression with schema  $(A_1, \dots, A_m, C_1, \dots, C_k, B_1, \dots, B_n)$  and is defined as the RA expression

$$\pi_{A_1, \dots, A_m, E_1.C_1, \dots, E_1.C_k, B_1, \dots, B_n}(E_1 \bowtie_C E_2)$$

where  $C$  is the condition

$$E_1.C_1 = E_2.C_1 \wedge \dots \wedge E_1.C_k = E_2.C_k$$

- Notice that if there are no common  $C$  attributes then  $E_1 \bowtie E_2 = E_1 \times E_2$ ; an empty conjunction is interpreted as true

## Natural joins (General case) in SQL

- If  $Q_{E_1}$  and  $Q_{E_2}$  denote the SQL queries corresponding to  $E_1$  and  $E_2$  then  $E_1 \bowtie E_2$  corresponds to the SQL query

```
SELECT A1, ..., Am, C1, ..., Ck, B1, ..., Bn
FROM   (QE1) e1 NATURAL JOIN (QE2) e2
```

## SQL as an algebra

- SQL is a declarative language.
- However, with the addition of the JOIN, NATURAL JOIN, and CROSS JOIN, **SQL can be seen as an algebra** that can faithfully simulate RA
- Indeed,
  - a WHERE clause can simulate a selection operation  $\sigma$ ;
  - a SELECT DISTINCT clause can simulate a projection operation  $\pi$ ;
  - a CROSS JOIN in the FROM clause can simulate a cartesian product; and
  - SQL has the UNION, INTERSECT, and EXCEPT operator that simulate  $\cup$ ,  $\cap$ , and  $-$

## Some observations about the power of the join operator

- Let  $E(A_1, \dots, A_m)$ ,  $E_1(A_1, \dots, A_m)$ , and  $E_2(A_1, \dots, A_m)$  be RA expressions with the same schema  $(A_1, \dots, A_m)$
- Then

$$\begin{aligned} E_1 \cap E_2 &= E_1 \bowtie E_2 \\ \sigma_{A_i \theta \mathbf{a}}(E) &= \pi_{E.*}(E \bowtie_{E.A_i \theta B} (B : \mathbf{a})) \\ \sigma_{A_i = A_j}(E) &= \pi_{E.*}(E \bowtie_{E.A_i = E.A_j} E') \end{aligned}$$

where  $E'$  is a copy of  $E$ .

- Conclusion: if we had taken  $\bowtie_C$  as a basic operation in RA, then  $\times$ ,  $\cap$ , and  $\sigma$  become derived operations.
- So all we need are the operations  $\pi$ ,  $\bowtie_C$ ,  $\cup$ , and  $-$ .
- In SQL, this means that we don't need the WHERE clause!

## Semijoins (Introduction)

- "Find the student information for each student who takes a course."

$$\pi_{S.sid, S.sname, S.age}(Student \bowtie Enroll)$$

- In this join expression, we "select" those tuples in the Student relation that **join with some** tuple in the Enroll relation.
- However, we are not interested in retaining any information (such as cno and grade) from this Enroll tuple.<sup>2</sup>
- This operation is called a **semijoin** (denoted  $\bowtie$ ) and we simply write

*Student  $\bowtie$  Enroll*

---

<sup>2</sup>Observe that  $\pi_{S.sid, S.sname, S.age}(Student \bowtie Enroll) \subseteq Student$

## Semijoin (Introduction)

Observe that

$$\pi_{S.sid, S.sname, S.age}(Student \bowtie Enroll) = Student \bowtie \pi_{E.sid}(Enroll)$$

- Notice that in the right side of this equation all the unnecessary information from the Enroll relation is projected out. Indeed, only its sid column is relevant in this join.
- We could have therefore also defined the semijoin as follows:

$$Student \ltimes Enroll = Student \bowtie \pi_{E.sid}(Enroll)$$

## Semijoins in SQL

- SQL does not have a semijoin operator. However semijoins can be simulated in various ways.
- (1) Using the **NATURAL JOIN** operator

```
SELECT sid, sname, age  
FROM Student NATURAL JOIN  
      (SELECT DISTINCT sid FROM Enroll) q
```

- (2) Using the **IN** predicate

```
SELECT sid, sname, age  
FROM Student  
WHERE sid IN (SELECT DISTINCT sid FROM Enroll)
```

## Semijoin (example)

- "Find the student information of students who take a course in the CS department."

$$\pi_{S.sid, S.sname, S.age}(Student \bowtie Enroll \bowtie \sigma_{C.Dept='CS'}(Course))$$

- With semijoins, this becomes

$$Student \ltimes (Enroll \ltimes \sigma_{C.Dept='CS'}(Course))$$

This expression can be evaluated in linear time using hashing!

- Be careful, **semijoin is not associative**:

$$Student \ltimes (Enroll \ltimes \sigma_{C.Dept='CS'}(Course)) \not\approx (Student \ltimes Enroll) \ltimes \sigma_{C.Dept='CS'}(Course)$$

## Semijoin (not associative)

- Be careful, **semijoin is not associative**:

$$Student \ltimes (Enroll \ltimes \sigma_{C.Dept='CS'}(Course)) \not\Leftarrow (Student \ltimes Enroll) \ltimes \sigma_{C.Dept='CS'}(Course)$$

- In fact,

$$(Student \ltimes Enroll) \ltimes \sigma_{C.Dept='CS'}(Course) \Leftrightarrow Student \ltimes Enroll$$

if  $\sigma_{C.dept='CS'}(Course) \neq \emptyset$ .

## Semijoin (General case)

- Let  $E_1(A_1, \dots, A_m, C_1, \dots, C_k)$  and  $E_2(C_1, \dots, C_k, B_1, \dots, B_n)$  be RA expressions
- Then

$$E_1 \ltimes E_2$$

is the **semijoin** between  $E_1$  and  $E_2$  (with schema  $(A_1, \dots, A_m, C_1, \dots, C_k)$ ) and it is defined as the following RA expression:

$$E_1 \bowtie \pi_{E_2.C_1, \dots, E_2.C_k}(E_2)$$

- $E_1 \ltimes E_2$  returns each tuples  $e_1$  from  $E_1$  wherefore there exists a tuples  $e_2$  of  $E_2$  such that  $e_1.C_1 = e_2.C_1 \wedge \dots \wedge e_1.C_k = e_2.C_k$ .
- So notice that one always has  $E_1 \ltimes E_2 \subseteq E_1$ .

## Semijoins (General case) in SQL

- If  $Q_{E_1}$  and  $Q_{E_2}$  denote the SQL queries corresponding to  $E_1$  and  $E_2$  then  $E_1 \ltimes E_2$  corresponds to the SQL query

```
SELECT e1.*
FROM (QE1) e1 NATURAL JOIN
      (SELECT DISTINCT C1, ..., Ck FROM (QE2) e2) q
```

- Alternatively,  $E_1 \ltimes E_2$  corresponds to the SQL query

```
SELECT e1.*
FROM (QE1) e1
WHERE (e1.C1, ..., e1.Ck) IN
      (SELECT DISTINCT e2.C1, ..., e2.Ck FROM (QE2)
```

## Semijoin (Time complexity)

- In general,  $E_1 \ltimes E_2$  can be implemented (using hashing) in linear time  $O(|E_1| + |E_2|)$ .
- By contrast,  $E_1 \bowtie E_2$  runs in the worst case in  $O(|E_1||E_2|)$ .
- Intuition by example:
  - Determining if a student takes a CS course can be done fast (linear time)
  - Determining for each student the CS courses taken by that student can be very expensive (quadratic time)
- Lesson: wherever possible, use semijoins

## Anti-semijoins

- Consider the query "Find the sid, name, and age of each student who is not enrolled in any course."
- In RA, this query can be formulated as follows

$\text{Student} - \text{Student} \times \text{Enroll}$ .

- This is correct since  $\text{Student} \times \text{Enroll}$  gives the students who are enrolled in some courses.
- Therefore, if we take the relation Student and subtract from it  $\text{Student} \times \text{Enroll}$ , we get indeed the students who are not enrolled in any course.

## Anti-semijoins

- "Find the sid, name, and age of each student who is not enrolled in any course". In RA,

$$Student - Student \times Enroll.$$

- There is an operation, called the **anti-semijoin**, and denoted by  $\overline{\times}$ , which is especially designed for such expressions.
- Using the **anti-semijoin** operation  $\overline{\times}$ , this query can be more succinctly expressed as

$$Student \overline{\times} Enroll.$$

## Anti-semijoins

- In general, given expressions  $E_1$  and  $E_2$ , the anti semijoin between  $E_1$  and  $E_2$ , i.e.,  $E_1 \setminus E_2$  is defined as the expression

$$E_1 - (E_1 \ltimes E_2)$$

- In SQL, the anti-semijoin can be supported using the **NOT IN** set predicate
- Just like the semijoin, the anti-semijoin can be implemented in linear time  $O(|E_1| + |E_2|)$  using hashing

## Anti-semijoins (General case) in SQL

- If  $Q_{E_1}$  and  $Q_{E_2}$  denote the SQL queries corresponding to  $E_1$  and  $E_2$ , then  $E_1 \setminus E_2$  corresponds to the SQL query

```
SELECT e1.*  
FROM (QE1)e1  
EXCEPT  
SELECT e1.*  
FROM (QE1) e1 NATURAL JOIN  
(SELECT DISTINCT C1,...,Ck FROM (QE2) e2)q
```

- Alternatively,  $E_1 \setminus E_2$  corresponds to the SQL query

```
SELECT e1.*  
FROM (QE1) e1  
WHERE (C1,...,Ck) NOT IN  
(SELECT DISTINCT C1,...,Ck FROM (QE2) e2)
```

## Semijoin and Anti-semijoins Special Cases

- Let  $E_1$  and  $E_2$  be expressions with the **same schemas**
- Then

$$E_1 \ltimes E_2 = E_1 \bowtie E_2 = E_1 \cap E_2$$

$$E_1 \overline{\ltimes} E_2 = E_1 - (E_1 \cap E_2) = E_1 - E_2$$

# Set Joins and Semijoins in Relational Algebra

Dirk Van Gucht<sup>1</sup>

<sup>1</sup>Indiana University

## Joins and semijoins

- Two types of joins
  - **Regular joins**: these are joins that compute relationships between objects ( $o_1, o_2$ ) based on **tuple component comparisons** (see the previous lecture on regular joins.)
  - **Set joins**: these are joins that compute relationships between objects ( $o_1, o_2$ ) based on **set comparisons between sets**  $S(o_1)$  and  $S(o_2)$  that are associated with objects  $o_1$  and  $o_2$ , respectively.
    - Think for example of finding pairs  $(s, p)$  such that student  $s$  only takes courses taught by professors.
    - If  $S(s)$  denotes the set of courses taken by  $s$  and  $S(p)$  denotes the set of courses taught by  $p$ , then we need to check if  $C(s) \subseteq C(p)$ .
- Analogously, there are also two types of semijoins: regular semijoins and set semijoins

## Set joins (Motivation)

- Consider the relations `Enroll(sid,cno)` and `TaughtBy(cno,pid)`.
- $(s, c)$  is in `Enroll` when student  $s$  is enrolled in course  $c$
- $(c, p)$  is in `TaughtBy` when course  $c$  is taught by professor  $p$
- We are interested in student-professor relationships  $(s, p)$  based on some complex join condition.
- "Find the student-professor pairs  $(s, p)$  such that student  $s$  takes [some|not only|not all|no|only|all|at least 2|... ] course(s) taught by professor  $p$ ."

## Set joins (Motivation)

- Consider the relations `Patient(pid, symptom)` and `Disease(sympton,disease)`.
- $(p, s)$  is in `Patient` when patient  $p$  exhibits health symptom  $s$
- $(s, d)$  is in `Disease` when  $s$  is a symptom of disease  $d$
- "Find the patient-disease pairs  $(p, d)$  such that patient  $p$  has [some|not only|not all|no|only|all|at least 2| $\dots$ ] symptoms associated with disease  $d$ ."

## Set **semijoins** (Motivation)

- Consider the relations  $\text{Patient}(\text{pid}, \text{sympton})$  and  $\text{Flu}(\text{sympton})$ .
- We are interested in finding patient that satisfy some complex condition.
- "Find each patient  $p$  such that  $p$  has  
[*some|not only|not all|no|only|all|at least 2|...|*] flu symptoms.

## Set **semijoins** (Advanced Google Search Engine)

- Consider the relations **Document(doc,word)** and the set of words  $\{Advanced, Database, Concept\}$ .
- A pair  $(d, w) \in Document$  when document  $d$  contains the word  $w$ .
- Advanced Google Search supports the following queries:  
"Find each document  $d$  such that  $d$  contains  
 $[some(any)|no|all]$  words in  
 $\{Advanced, Database, Concepts\}$ .
- You can try this out for the  $all$  query and you will see documents that refer to B561.
- [https://www.google.com/advanced\\_search](https://www.google.com/advanced_search)
- So set semijoins are frequently used in real search applications

## Set joins and semijoin in RA (Objective)

- Derive relational algebra expressions for set joins and semijoins that supports queries with quantifiers.
- The technique we use for this follows the Venn-diagram method used in the lecture on "Queries with Quantifiers" to derive SQL queries for such queries.
- We will discuss how the derived RA expressions provide deep insight into the time and space complexities for different set joins
- Note: this is a very technical lecture that relies on understanding of logic and set theory

## Set joins in RA (Introduction-Terminology)

We begin by introducing some terminology and state our assumptions.

- Denote by  $\mathcal{S}$  the set of all student sids
- Denote by  $\mathcal{P}$  the set of all professor pids
- Denote by  $\mathcal{C}$  the set of all course cnos
- Denote by  $E$  the relation *Enroll*
- Denote by  $T$  the relation *TaughtBy*
- We will assume that the natural foreign key constraints hold:

$$\begin{aligned}\pi_{sid}(E) &\subseteq \mathcal{S} \\ \pi_{cno}(E) &\subseteq \mathcal{C} \\ \pi_{cno}(T) &\subseteq \mathcal{C} \\ \pi_{pid}(T) &\subseteq \mathcal{P}\end{aligned}$$

## Set joins in RA (Parameterized sets)

- Let  $s$  be a student sid (i.e.,  $s \in S$ ) and let  $p$  be a professor pid (i.e.,  $p \in P$ ).
- We will consider **sets parameterized** by  $s$  and  $p$ .
- Both these sets are subsets of  $C$ , i.e., the set of cnos of courses.
  - $E(s)$  is the set of cnos of courses in which student  $s$  is enrolled.<sup>1</sup>

$$E(s) = \{c \mid (s, c) \text{ is a tuple in the Enroll relation}\}$$

- $T(p)$  is the set of cnos of courses taught by professor  $p$ .<sup>2</sup>

$$T(p) = \{c \mid (c, p) \text{ is a tuple in the TaughtBy relation}\}$$

---

<sup>1</sup>If  $s$  is not enrolled in any course then  $E(s) = \emptyset$ .

<sup>2</sup>If  $p$  does not teach any course then  $T(p) = \emptyset$ .

## Set joins (Regions)

- For each student-professor pair  $(s, p)$ , the sets  $E(s)$  and  $T(p)$  define 4 pairwise disjoint "regions" in the Venn diagram associated with these sets.<sup>3</sup>
- These regions partition the set of courses  $\mathcal{C}$ :

$E(s) \cap T(p)$  = the courses **enrolling** student  $s$  and  
**taught by** professor  $p$

$E(s) - T(p)$  = the courses **enrolling** student  $s$  but  
**not taught by** professor  $p$

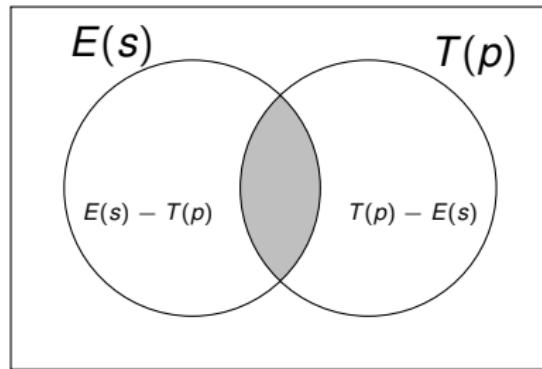
$T(p) - E(s)$  = the courses **not enrolling** student  $s$  but  
**taught by** professor  $p$

$\mathcal{C} - (E(s) \cup T(p))$  = the courses **not enrolling** student  $s$  and  
**not taught by** professor  $p$

---

<sup>3</sup>It is recommended that you revisit the lecture of "Queries with Quantifiers" to further understand this Venn diagram.

## Venn diagram and regions for $E(s)$ and $T(p)$



The gray region is  $E(s) \cap T(p)$

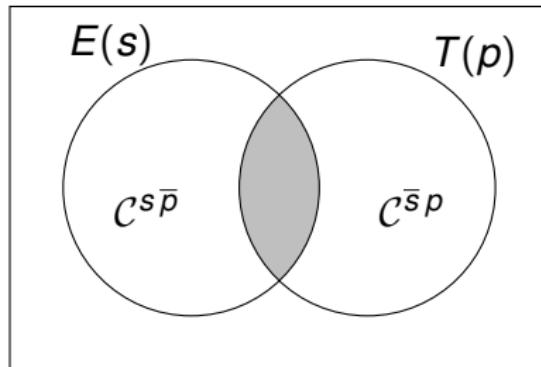
The region outside both  $E(s)$  and  $T(p)$  is  $\mathcal{C} - (E(s) \cup T(p))$

## Set joins (Regions)

- For each student-professor pair  $(s, p)$ , the sets  $E(s)$  and  $T(p)$  define 4 pairwise disjoint "regions" that partition the set of cnos of courses  $\mathcal{C}$ :
- We use the following notations to denote these regions:

| Region                           | Notation                       |
|----------------------------------|--------------------------------|
| $E(s) \cap T(p)$                 | $\mathcal{C}^{sp}$             |
| $E(s) - T(p)$                    | $\mathcal{C}^{s\bar{p}}$       |
| $T(p) - E(s)$                    | $\mathcal{C}^{\bar{s}p}$       |
| $\mathcal{C} - (E(s) \cup T(p))$ | $\mathcal{C}^{\bar{s}\bar{p}}$ |

## Venn diagram and regions for $E(s)$ and $T(p)$



The gray region is  $C^{sp}$

The region outside both  $E(s)$  and  $T(p)$  is  $C^{\bar{s}\bar{p}}$ . (From now on, we will ignore this region in the lecture.)

## Set joins (Region analysis)

- Consider again the regions  $\mathcal{C}^{sp}$ ,  $\mathcal{C}^{s\bar{p}}$ , and  $\mathcal{C}^{\bar{s}p}$ .
- These regions can be either not-empty or empty.
- Depending on this, we can distinguish 6 set joins. They correspond to the **some**, **no**, **not only**, **only**, **not all**, and **all** quantifiers we considered earlier:

$$\mathcal{C}^{sp} \neq \emptyset \leftrightarrow \text{some}$$

$$\mathcal{C}^{sp} = \emptyset \leftrightarrow \text{no}$$

$$\mathcal{C}^{s\bar{p}} \neq \emptyset \leftrightarrow \text{not only}$$

$$\mathcal{C}^{s\bar{p}} = \emptyset \leftrightarrow \text{only}$$

$$\mathcal{C}^{\bar{s}p} \neq \emptyset \leftrightarrow \text{not all}$$

$$\mathcal{C}^{\bar{s}p} = \emptyset \leftrightarrow \text{all}$$

- Once the analysis of these cases is understood, it becomes straightforward to study a very large set of other interesting (and practical) set joins.

## Set join (not-empty cases)

- We begin with the set joins for **some**, **not only**, and **not all**. I.e., those corresponding to **non-empty** regions.
- To perform our analysis, it helps to formulate precisely the queries we want to express as RA expressions:

$$\begin{array}{lll} \text{some} & \leftrightarrow & \{(s, p) | \exists c : c \in \mathcal{C}^{sp}\} \\ & \leftrightarrow & \{(s, p) | \exists c : c \in E(s) \cap T(p)\} \\ & \leftrightarrow & \{(s, p) | \exists c : c \in E(s) \wedge c \in T(p)\} \\ \text{not only} & \leftrightarrow & \{(s, p) | \exists c : c \in \mathcal{C}^{s\bar{p}}\} \\ & \leftrightarrow & \{(s, p) | \exists c : c \in (E(s) - T(p))\} \\ & \leftrightarrow & \{(s, p) | \exists c : c \in E(s) \wedge c \notin T(p)\} \\ \text{not all} & \leftrightarrow & \{(s, p) | \exists c : c \in \mathcal{C}^{\bar{s}p}\} \\ & \leftrightarrow & \{(s, p) | \exists c : c \in (T(p) - E(s))\} \\ & \leftrightarrow & \{(s, p) | \exists c : c \in T(p) \wedge c \notin E(s)\} \end{array}$$

## Set join (**some** case)

- Recall

$$\begin{aligned}\text{some} &\leftrightarrow \{(s, p) | \exists c : c \in C^{sp}\} \\ &\leftrightarrow \{(s, p) | \exists c : c \in E(s) \wedge c \in T(p)\} \\ &\leftrightarrow \{(s, p) | \exists c : E(s, c) \wedge T(c, p)\}\end{aligned}$$

- We first need to find an RA expression for the set

$$\{(s, c, p) | E(s, c) \wedge T(c, p)\}.$$

- But this is easy since, by the definition of the natural join  $\bowtie$

$$\{(s, c, p) | E(s, c) \wedge T(c, p)\} = E \bowtie T.$$

Consequently, the RA expression for **some** is

$$\pi_{sid,pid}(E \bowtie T)$$

## Set join (**not only** case)

- Recall

$$\begin{aligned}\text{not only} &\leftrightarrow \{(s, p) | \exists c : c \in \mathcal{C}^{s \bar{p}}\} \\ &\leftrightarrow \{(s, p) | \exists c : c \in E(s) \wedge \neg(c \in T(p))\} \\ &\leftrightarrow \{(s, p) | \exists c : E(s, c) \wedge \neg T(c, p)\}\end{aligned}$$

- We first need to find an RA expression for the set

$$\{(s, c, p) | E(s, c) \wedge \neg T(c, p)\}.$$

$$\begin{aligned}\{(s, c, p) | E(s, c) \wedge \neg T(c, p)\} &= \{(s, c, p) | (E(s, c) \wedge p \in \mathcal{P}) \wedge \neg(E(s, c) \wedge T(c, p))\} \quad (1) \\ &= \{(s, c, p) | (s, c, p) \in (E \times \mathcal{P}) \wedge \neg((s, c, p) \in E \bowtie T)\} \\ &= (E \times \mathcal{P}) - (E \bowtie T)\end{aligned}$$

The equality (1) will be shown on the next slides.  
Consequently, the RA expression for **not only** is

$$\pi_{sid,pid}((E \times \mathcal{P}) - (E \bowtie T))$$

## Set join (**not only** case using natural join)

- To prove equality (1), we need to show that the following statements are logically equivalent

$$E(s, c) \wedge \neg T(c, p)$$

$$(E(s, c) \wedge p \in \mathcal{P}) \wedge \neg(E(s, c) \wedge T(c, p))$$

Since, clearly  $p \in \mathcal{P}$  is true, we have to show that the following statements are equivalent:<sup>4</sup>

$$E(s, c) \wedge \neg T(c, p)$$

$$E(s, c) \wedge \neg(E(s, c) \wedge T(c, p))$$

---

<sup>4</sup>We are assuming that there is at least one professor.

## Set join (**not only** case using natural join)

We need to show that the following statements are equivalent:

$$E(s, c) \wedge \neg T(c, p) \quad (\text{I})$$

$$E(s, c) \wedge \neg(E(s, c) \wedge T(c, p)) \quad (\text{II})$$

To see this, you need to determine if statements (I) and (II) have the same truth values. This is the case as shown in the following truth table:

| $E(s, c)$ | $T(c, p)$ | Statement (I) | Statement (II) |
|-----------|-----------|---------------|----------------|
| true      | true      | false         | false          |
| true      | false     | true          | true           |
| false     | true      | false         | false          |
| false     | false     | false         | false          |

## Set join (**not all** case)

- Recall

$$\begin{aligned}\text{not all} &\leftrightarrow \{(s, p) | \exists c : c \in \mathcal{C}^{\bar{s}p}\} \\ &\leftrightarrow \{(s, p) | \exists c : c \in T(p) \wedge \neg(c \in E(s))\} \\ &\leftrightarrow \{(s, p) | \exists c : T(c, p) \wedge \neg E(s, c)\}\end{aligned}$$

- We first need to find an RA expression for the set

$$\{(s, c, p) | T(c, p) \wedge \neg E(s, c)\}.$$

$$\begin{aligned}\{(s, c, p) | T(c, p) \wedge \neg E(s, c)\} &= \{(s, c, p) | (s \in \mathcal{S} \wedge T(c, p)) \wedge \neg(E(s, c) \wedge T(c, p))\} \\ &= \{(s, c, p) | (s, c, p) \in (\mathcal{S} \times T) \wedge \neg((s, c, p) \in E \bowtie T)\} \\ &= (\mathcal{S} \times T) - (E \bowtie T)\end{aligned}\tag{1}$$

The equality (1) will be shown on the next slides.  
Consequently, the RA expression for **not all** is

$$\pi_{sid,pid}((\mathcal{S} \times T) - (E \bowtie T))$$

## Set join (**not all** case using natural join)

- To prove equality (1), we need to show that the following statements are logically equivalent

$$T(c, p)) \wedge \neg E(s, c)$$

$$(s \in S \wedge T(c, p)) \wedge \neg(E(s, c) \wedge T(c, p))$$

Since, clearly,  $s \in S$  is true, we need to show that the following statements are equivalent:<sup>5</sup>

$$T(c, p) \wedge \neg E(s, c)$$

$$T(c, p) \wedge \neg(E(s, c) \wedge T(c, p))$$

---

<sup>5</sup>We assume that there is at least one student.

## Set join (**not all** case using natural join)

We need to show that the following statements are equivalent:

$$T(c, p) \wedge \neg E(s, c) \quad (\text{I})$$

$$T(c, p) \wedge \neg(E(s, c) \wedge T(c, p)) \quad (\text{II})$$

To see this, you need to determine if statements (I) and (II) have the same truth values. This is the case as shown in the following truth table:

| $E(s, c)$ | $T(c, p)$ | Statement (I) | Statement (II) |
|-----------|-----------|---------------|----------------|
| true      | true      | false         | false          |
| true      | false     | false         | false          |
| false     | true      | true          | true           |
| false     | false     | false         | false          |

## Set joins (not-empty cases)

- We have now developed the RA expressions for the following set joins:

$$\text{some} = \pi_{sid,pid}(E \bowtie T)$$

$$\text{not only} = \pi_{sid,pid}((E \times \mathcal{P}) - (E \bowtie T))$$

$$\text{not all} = \pi_{sid,pid}((S \times T) - (E \bowtie T))$$

- Notice that the time and space complexities for *not only* and *not all* are at least  $O(|E| * |\mathcal{P}|)$  and  $O(|T| * |\mathcal{S}|)$ , respectively. (At IU,  $160000 * 2000$  and  $6000 * 40000$ .)
- These set joins can be very expensive to evaluate.
- Observe that the time and space complexity for  $E \bowtie S$  is never more than  $O(|E| + |S| + |S| * |\mathcal{P}|)$ <sup>6</sup>

---

<sup>6</sup>If we use the hash join algorithm.

## Set joins (generalized intersection and difference)

A more compact way to remember these expressions is by using the following notations:

$$\begin{aligned} E \sqcap T &\leftrightarrow E \bowtie T \\ E \ominus T &\leftrightarrow (E \times \mathcal{P}) - (E \bowtie T) \\ T \ominus E &\leftrightarrow (\mathcal{S} \times T) - (E \bowtie T) \end{aligned}$$

With these notations, we get

$$\begin{aligned} \text{some} &= \pi_{sid,pid}(E \sqcap T) \\ \text{not only} &= \pi_{sid,pid}(E \ominus T) \\ \text{not all} &= \pi_{sid,pid}(T \ominus E) \end{aligned}$$

We call  $\sqcap$  the generalized intersection and  $\ominus$  the generalized difference.

## Set joins (empty cases)

- Finding RA expressions for the empty cases is now easy: they consist of subtracting the corresponding not-empty cases from the set of all possible student-professor pairs  $(s, p)$ , i.e., the set  $\mathcal{S} \times \mathcal{P}$ .
- So, we get the following RA expressions for the set joins associated with **no**, **not only**, and **all**:

$$\begin{aligned}\text{no} &= \mathcal{S} \times \mathcal{P} - (\pi_{sid,pid}(E \sqcap T)) \\ \text{only} &= \mathcal{S} \times \mathcal{P} - (\pi_{sid,pid}(E \ominus T)) \\ \text{all} &= \mathcal{S} \times \mathcal{P} - (\pi_{sid,pid}(T \ominus E))\end{aligned}$$

## Set joins (Summary)

The following are then the RA expressions for all of the cases considered:

|          |   |                                                                 |
|----------|---|-----------------------------------------------------------------|
| some     | = | $\pi_{sid,pid}(E \sqcap T)$                                     |
| not only | = | $\pi_{sid,pid}(E \ominus T)$                                    |
| not all  | = | $\pi_{sid,pid}(T \ominus E)$                                    |
| no       | = | $\mathcal{S} \times \mathcal{P} - (\pi_{sid,pid}(E \sqcap T))$  |
| only     | = | $\mathcal{S} \times \mathcal{P} - (\pi_{sid,pid}(E \ominus T))$ |
| all      | = | $\mathcal{S} \times \mathcal{P} - (\pi_{sid,pid}(T \ominus E))$ |

## Set joins **some** and **no** in SQL

- **some**

```
SELECT DISTINCT sid, pid  
FROM Enroll NATURAL JOIN TaughtBy
```

- **no**

```
SELECT sid, pid  
FROM Student CROSS JOIN Professor  
EXCEPT  
SELECT DISTINCT sid, pid  
FROM Enroll NATURAL JOIN TaughtBy
```

## Set joins **not only** and **only** in SQL

**not only**

```
SELECT sid, pid
FROM   (SELECT sid, cno, pid
        FROM Enroll CROSS JOIN Professor
EXCEPT
        SELECT sid, cno, pid
        FROM Enroll NATURAL JOIN TaughtBy) q
```

**only**

```
SELECT sid, pid
FROM Student      CROSS JOIN Professor
EXCEPT
SELECT sid, pid
FROM   (SELECT sid, cno, pid
        FROM Enroll CROSS JOIN Professor
EXCEPT
        SELECT sid, cno, pid
        FROM Enroll NATURAL JOIN TaughtBy) q
```

## Set joins **not all** and **all** in SQL

not all

```
SELECT sid, pid
FROM   (SELECT sid, cno, pid
        FROM Student CROSS JOIN TaughtBy
EXCEPT
        SELECT sid, cno, pid
        FROM Enroll NATURAL JOIN TaughtBy) q
```

all

```
SELECT sid, pid
FROM   Student      CROSS JOIN Professor
EXCEPT
SELECT sid, pid
FROM   (SELECT sid, cno, pid
        FROM Student CROSS JOIN TaughtBy
EXCEPT
        SELECT sid, cno, pid
        FROM Enroll NATURAL JOIN TaughtBy) q
```

The queries run in  $O(|Student| * |TaughtBy| + |Enroll| * |TaughtBy|)$

## Set joins **all** in Pure SQL

```
SELECT      sid, pid
FROM        Student
EXCEPT
SELECT      sid, pid
FROM        (SELECT      sid, cno, pid
            FROM        Student
            CROSS JOIN TaughtBy
EXCEPT
SELECT      sid, cno, pid
FROM        Enroll
NATURAL JOIN TaughtBy q
```

Contrast this with the **all** set join expressed in Pure SQL

```
SELECT      sid, pid
FROM        Student s, Professor p
WHERE      NOT EXISTS (SELECT 1
                        FROM TaughtBy t
                        WHERE t.pid = p.pid AND
                            t.cno NOT IN (SELECT e.cno
   FROM Enroll e
   WHERE e.sid = s.sid))
```

This query runs in  $O(|Student| * |Professor| * |TaughtBy| * |Enroll|)$  and is thus orders of magnitude slower than the above RA SQL query which runs in  $O(|Student| * |TaughtBy| + |Enroll| * |TaughtBy|)$ .

## Set joins (Example)

Consider the following queries:

"Find the student sid pairs  $(s_1, s_2)$  such that student  $s_1$  takes  
[some|not only|not all|no|only|all] course(s) that student  $s_2$   
takes."

The queries are very similar to those we have been  
considering.

The only difference is that we now have to involve the *Enroll*  
relation twice: ones for student  $s_1$  and the other for student  $s_2$ .

## Set joins (Example)

Let us denote by  $\tilde{E}$  the RA expression  $\rho_{sid \rightarrow \widetilde{sid}}(\pi_{cno, sid}(E))$ . So  $\tilde{E}$  is the same as  $E$ , except that its schema is  $(cno, \widetilde{sid})$  instead of  $(sid, cno)$ . We have renamed  $sid$  by  $\widetilde{sid}$ .

For this example, the **search space** is the set of all student pairs  $\mathcal{S} \times \mathcal{S}$  and we assume its schema is  $(\widetilde{sid}, \widetilde{sid})$ .

Then the RA expressions for the above six queries are

|                 |     |                                                                                      |
|-----------------|-----|--------------------------------------------------------------------------------------|
| <b>some</b>     | $=$ | $\pi_{sid, \widetilde{sid}}(E \bowtie \tilde{E})$                                    |
| <b>not only</b> | $=$ | $\pi_{sid, \widetilde{sid}}(E \ominus \tilde{E})$                                    |
| <b>not all</b>  | $=$ | $\pi_{sid, \widetilde{sid}}(\tilde{E} \ominus E)$                                    |
| <b>no</b>       | $=$ | $\mathcal{S} \times \mathcal{S} - (\pi_{sid, \widetilde{sid}}(E \bowtie \tilde{E}))$ |
| <b>only</b>     | $=$ | $\mathcal{S} \times \mathcal{S} - (\pi_{sid, \widetilde{sid}}(E \ominus \tilde{E}))$ |
| <b>all</b>      | $=$ | $\mathcal{S} \times \mathcal{S} - (\pi_{sid, \widetilde{sid}}(\tilde{E} \ominus E))$ |

## More set joins

- The RA region expressions  $E \sqcap T$ ,  $E \ominus T$ , and  $T \ominus E$  permit us to express many more set joins. We will consider one such example next.
- "Find the student-professor pairs  $(s, p)$  such that student  $s$  takes at least 2 courses not taught by professor  $p$ ."
- More formally, we need to find a RA expression for the query

$$\{(s, p) \mid \exists c_1, c_2 : c_1 \neq c_2 \wedge c_1 \in (E(s) - T(p)) \wedge c_2 \in (E(s) - T(p))\}$$

- Alternatively,

$$\{(s, p) \mid \exists c_1, c_2 : c_1 \neq c_2 \wedge \{c_1, c_2\} \subseteq (E(s) - T(p))\}$$

## More set joins

- "Find the student-professor pairs  $(s, p)$  such that student  $s$  takes at least 2 courses not taught by professor  $p$ ."

$$\{(s, p) \mid \exists c_1, c_2 : c_1 \neq c_2 \wedge \{c_1, c_2\} \subseteq (E(s) - T(p))\}$$

- From this formulation, it is clear that we need to involve the region expression  $E \ominus T$ , but we need two copies of it, one to reason about  $c_1$  and the other to reason about  $c_2$ .
- Let these two copies be  $E_1 \ominus T_1$  with schema  $(sid, cno_1, pid)$  and  $E_2 \ominus T_2$  with schema  $(sid, cno_2, pid)$ .
- Then the RA expression for the query is as follows:

$$\pi_{sid,pid}(\sigma_{cno_1 \neq cno_2}((E_1 \ominus T_1) \bowtie (E_2 \ominus T_2)))$$

## Set semijoins (Motivation)

- Consider the relations  $\text{Enroll}(\text{sid}, \text{cno})$  and the relation of CS courses  $\text{CS}(\text{cno})$ .
- We are interested in students satisfying some property.
- "Find each student sid  $s$  such that student  $s$  takes [some|not only|not all|no|only|all] CS course(s)."

## Set semijoins (Solution)

An analysis very similar to that for set joins gives the following RA queries for these queries.

First observe that  $E \bowtie CS = E \ltimes CS$ .

|          |   |                                 |   |                           |
|----------|---|---------------------------------|---|---------------------------|
| some     | = | $\pi_{sid}(E \bowtie CS)$       | = | $\pi_{sid}(E \ltimes CS)$ |
| not only | = | $\pi_{sid}(E \ominus CS)$       |   |                           |
| not all  | = | $\pi_{sid}(CS \ominus E)$       |   |                           |
| no       | = | $S - (\pi_{sid}(E \ltimes CS))$ |   |                           |
| only     | = | $S - (\pi_{sid}(E \ominus CS))$ |   |                           |
| all      | = | $S - (\pi_{sid}(CS \ominus E))$ |   |                           |

Where  $E \ominus CS = E - (E \ltimes CS) = E \overline{\ltimes} CS$

$$CS \ominus E = (S \times CS) - (E \ltimes CS)$$

## Set semijoins (Efficiency consideration)

Reconsider the 6 semijoins:

$$\begin{aligned}\text{some} &= \pi_{sid}(E \ltimes CS) \\ \text{not only} &= \pi_{sid}(E \overline{\ltimes} CS)) \\ \text{not all} &= \pi_{sid}((\pi_{sid}(S) \times CS) - (E \ltimes CS)) \\ \text{no} &= \pi_{sid}(S) - (\pi_{sid}(E \ltimes CS)) \\ \text{only} &= \pi_{sid}(S) - \pi_{sid}(E \overline{\ltimes} CS) \\ \text{all} &= \pi_{sid}(S) - \pi_{sid}((\pi_{sid}(S) \times CS) - (E \ltimes CS))\end{aligned}$$

Let  $k = |S|$ ,  $l = |CS|$ , and  $n = |E|$ .

- Then, using hashing methods, the **some**, **no**, **not only**, **only** set semijoins can be implemented in **linear time and space**  $O(|k| + |l| + |n|)$ . (I.e., extremely fast.)
- However, the **not all** and **all** set semijoins require **quadratic time and space**  $O((k * l) + n)$  because of the need to compute  $\pi_{sid}(S) \times CS$ . (I.e., can be extremely expensive.)

## Set semijoin **some** and **no** in SQL

- **some**

```
SELECT DISTINCT sid
FROM Enroll NATURAL JOIN CS
```

- **no**

```
SELECT sid
FROM Student
EXCEPT
SELECT DISTINCT sid
FROM Enroll NATURAL JOIN CS
```

## Set semijoin **not only** in SQL

- **not only**

```
SELECT DISTINCT sid
  FROM (SELECT sid, cno
        FROM Enroll
      EXCEPT
      SELECT sid, cno
        FROM Enroll NATURAL JOIN CS) q
```

## Set semijoin **only** SQL

- **only**

```
SELECT sid
FROM Student
EXCEPT
SELECT DISTINCT sid
FROM (SELECT sid, cno
      FROM Enroll
EXCEPT
      SELECT sid, cno
      FROM Enroll NATURAL JOIN CS) q
```

## Set semijoin **not all** in SQL

- **not all**

```
SELECT DISTINCT sid
  FROM (SELECT sid, cno
         FROM (SELECT sid from Student) s CROSS JOIN CS
       EXCEPT
      SELECT sid, cno
        FROM Enroll NATURAL JOIN CS) q
```

## Set semijoin **all** in SQL

- **all**

```
SELECT    sid
FROM      Student s
EXCEPT
SELECT    DISTINCT sid
FROM      (SELECT sid, cno
          FROM  (SELECT sid from Student) s CROSS JOIN CS
EXCEPT
SELECT    sid, cno
FROM      Enroll NATURAL JOIN CS) q
```

## Set joins (General case)

- Let  $E_1(A_1, \dots, A_m, C_1, \dots, C_k)$  and  $E_2(C_1, \dots, C_k, B_1, \dots, B_n)$  be RA expressions.
- We assume that each attribute has an associated domain: these are denoted  $\mathcal{A}_1, \dots, \mathcal{A}_m, \mathcal{C}_1, \dots, \mathcal{C}_k, \mathcal{B}_1, \dots, \mathcal{B}_n$ .
- We can then build RA expressions  $E_1 \sqcap E_2$ ,  $E_1 \ominus E_2$ , and  $E_2 \ominus E_1$  as follows:

$$E_1 \sqcap E_2 = E_1 \bowtie E_2$$

$$E_1 \ominus E_2 = (E_1 \times \mathcal{B}_1 \times \dots \times \mathcal{B}_n) - (E_1 \bowtie E_2)$$

$$E_2 \ominus E_1 = (\mathcal{A}_1 \times \dots \times \mathcal{A}_m \times E_2) - (E_1 \bowtie E_2)$$

- Using these region expressions we can build numerous other set-joins.

## Set joins (From general case to special cases)

$$E_1 \sqcap E_2 = E_1 \bowtie E_2$$

$$E_1 \ominus E_2 = (E_1 \times \mathcal{B}_1 \times \cdots \times \mathcal{B}_n) - (E_1 \bowtie E_2)$$

$$E_2 \ominus E_1 = (\mathcal{A}_1 \times \cdots \times \mathcal{A}_m \times E_2) - (E_1 \bowtie E_2)$$

We can now consider special cases:

- When  $m = k = n = 1$  we have the region expressions for the set joins we have seen above.
- When  $m = k = 1$  and  $n = 0$  we have the region expressions for semi set joins we have seen above.
- When  $m = n = 0$ , then

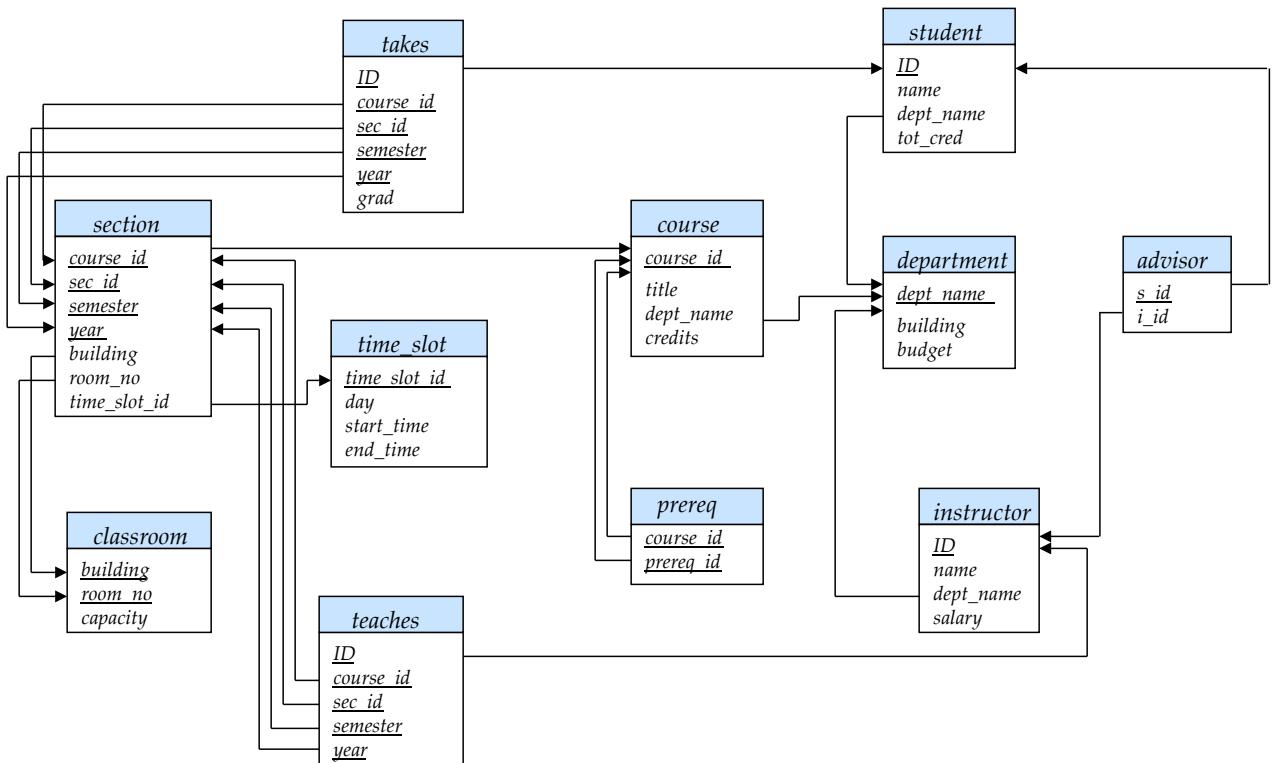
$$E_1 \sqcap E_2 = E_1 \cap E_2$$

$$E_1 \ominus E_2 = E_1 - E_2$$

$$E_2 \ominus E_1 = E_2 - E_1$$

## 1 Database Schema Diagrams

We will use the university sample relational schema defined in lectures. The following schema diagram illustrates all the relations in the database. Each relation has primary keys underlined. Draw arrows to show all *foreign key* constraints.



## 2 Relational Algebra

We use relational algebra to specify *queries* on a database. This is the formal mathematical notational. Later we will see how this translates into SQL.

Summary of Relational Algebra

---

<sup>1</sup>Diagrams and examples taken from Database System Concepts 6th Edition

| Symbol (Name)                   | Example of Use                                                                                                                                                                      |
|---------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\sigma$<br>(Selection)         | $\sigma_{\text{salary} >= 85000}(\text{instructor})$<br>Return rows of the input relation that satisfy the predicate.                                                               |
| $\Pi$<br>(Projection)           | $\Pi_{ID, \text{salary}}(\text{instructor})$<br>Output specified attributes from all rows of the input relation. Remove duplicate tuples from the output.                           |
| $\bowtie$<br>(Natural join)     | $\text{instructor} \bowtie \text{department}$<br>Output pairs of rows from the two input relations that have the same value on all attributes that have the same name.              |
| $\times$<br>(Cartesian product) | $\text{instructor} \times \text{department}$<br>Output all pairs of rows from the two input relations (regardless of whether or not they have the same values on common attributes) |
| $\cup$<br>(Union)               | $\Pi_{name}(\text{instructor}) \cup \Pi_{name}(\text{student})$<br>Output the union of tuples from the two input relations.                                                         |

## 2.1 Lets practice...

The university schema and relations are at the end of this handout.

1. What is the result of  $\sigma_{s\_id=ID}(\text{student} \times \text{advisor})$ ?
2. Explain what each of the following expressions do:
  - (a)  $\sigma_{year \geq 2009}(\text{takes}) \bowtie \text{student}$
  - (b)  $\sigma_{year \geq 2009}(\text{takes} \bowtie \text{student})$
  - (c)  $\Pi_{ID, \text{name}, \text{course\_id}}(\text{student} \bowtie \text{takes})$
3. Suppose we have the following relational database:

*employee (person\_name, street, city)  
 works(person\_name, company\_name, salary)  
 company (company\_name, city)*

- (a) Find the names of all employees who live in city ‘Miami’.

$$\Pi_{\text{person\_name}}(\sigma_{\text{city}='Miami'}(\text{employee}))$$

- (b) Find the names of all employees whose salary is greater than \$100,000.

$$\Pi_{\text{person\_name}}(\sigma_{\text{salary} > 100000}(\text{works}))$$

- (c) Find the names of all employees who live in ‘Miami’ and whose salary is greater than \$100,000.

$$\Pi_{person\_name}(\sigma_{city='Miami'} \wedge salary > 100000(employee \bowtie works))$$

- (d) Find the names of all employees who work for “First Bank Corporation”.

$$\Pi_{person\_name}(\sigma_{company\_name='First Bank Corporation'}(works))$$

- (e) Find the names and cities of residence of all employees who work for “First Bank Corporation”.

$$\Pi_{person\_name, city}(\sigma_{company\_name='First Bank Corporation'}(works \bowtie employee))$$

- (f) Find the names, street address, and cities of residence of all employees who work for “First Bank Corporation” and earn more than \$10,000.

$$\Pi_{person\_name, street, city}(\sigma_{company\_name='First Bank Corporation'} \wedge salary > 10000(employee \bowtie works))$$

4. Suppose we have the following relational database:

```

branch(branch_name, branch_city, assets)
customer(customer_name, customer_street, customer_city)
loan(loan_number, branch_name, amount)
borrower(customer_name, loan_number)
account(account_number, branch_name, balance)
depositor(customer_name, account_number)

```

- (a) Underline appropriate primary keys.

Primary keys are bold font. Note that some of this is up to interpretation of the database - feel free to discuss and come to some agreement. A sample solution might be:

```

branch(branch_name, branch_city, assets)
customer(customer_name, customer_street, customer_city) ** customer_names may
not be unique**
loan(loan_number, branch_name, amount) ** assume that loan numbers are across all
branches
borrower(customer_name, loan_number)
account(account_number, branch_name, balance)
depositor(customer_name, account_number)

```

- (b) Given your choice of primary keys, identify appropriate foreign keys. Draw a relation diagram to indicate foreign keys.

try having the students draw a box for each schema and then draw the relation diagram for the foreign keys as is done on the diagram on the first page.

$\text{branch}(\text{branch\_name}, \text{branch\_city}, \text{assets})$   
 $\text{customer}(\text{customer\_name}, \text{customer\_street}, \text{customer\_city}) \text{ ** names may not be unique**}$   
 $\text{loan}(\text{loan\_number}, \text{branch\_name}, \text{amount}) \text{ ** assume that loan numbers are across all branches}$   
 $\text{borrower}(\text{customer\_name}, \text{loan\_number})$   
 $\text{account}(\text{account\_number}, \text{branch\_name}, \text{balance})$   
 $\text{depositor}(\text{customer\_name}, \text{account\_number})$

Foreign Keys:

$\text{loan}$  and  $\text{account}$  have foreign key  $\text{branch\_name}$  for  $\text{branch}$ .

$\text{borrower}$  has foreign key  $\text{loan\_number}$  for  $\text{loan}$ .

$\text{depositor}$  has foreign key  $\text{account\_number}$  for  $\text{account}$ .

- (c) Find the names of all branches located in ‘Chicago’.

$\Pi_{\text{branch\_name}}(\sigma_{\text{branch\_city}=\text{'Chicago'}}(\text{branch}))$

- (d) Find the names of all borrowers who have a loan in branch ‘Down-town’.

$\Pi_{\text{customer\_name}}(\sigma_{\text{branch\_name}=\text{'Down-town'}}(\text{borrower} \bowtie \text{loan}))$

- (e) Find all loan numbers with a loan value greater than \$10, 000.

$\Pi_{\text{loan\_number}}(\sigma_{\text{amount}>10000}(\text{loan}))$

- (f) Find the names of all depositors who have an account with a value greater than \$6000.

$\Pi_{\text{customer\_name}}(\sigma_{\text{balance}>6000}(\text{depositor} \bowtie \text{account}))$

### 3 University Relations

Relations and their schemas:

$\text{classroom}(\text{building}, \underline{\text{room\_number}}, \text{capacity})$   
 $\text{department}(\underline{\text{dept\_name}}, \text{building}, \text{budget})$   
 $\text{course}(\underline{\text{course\_id}}, \underline{\text{title}}, \underline{\text{dept\_name}}, \text{credits})$   
 $\text{instructor}(\underline{\text{ID}}, \underline{\text{name}}, \underline{\text{dept\_name}}, \text{salary})$   
 $\text{section}(\underline{\text{course\_id}}, \underline{\text{sec\_id}}, \underline{\text{semester}}, \underline{\text{year}}, \text{building}, \underline{\text{room\_number}}, \underline{\text{time\_slot\_id}})$   
 $\text{teaches}(\underline{\text{ID}}, \underline{\text{course\_id}}, \underline{\text{sec\_id}}, \underline{\text{semester}}, \underline{\text{year}})$   
 $\text{student}(\underline{\text{ID}}, \underline{\text{name}}, \underline{\text{dept\_name}}, \text{tot\_cred})$   
 $\text{takes}(\underline{\text{ID}}, \underline{\text{course\_id}}, \underline{\text{sec\_id}}, \underline{\text{semester}}, \underline{\text{year}}, \text{grade})$   
 $\text{advisor}(\underline{s\_ID}, \underline{i\_ID})$   
 $\text{time\_slot}(\underline{\text{time\_slot\_id}}, \underline{\text{day}}, \underline{\text{start\_time}}, \underline{\text{end\_time}})$   
 $\text{prereq}(\underline{\text{course\_id}}, \underline{\text{prereq\_id}})$

| ID    | course_id | sec_id | semester | year | ID    | name       | dept_name  | salary |
|-------|-----------|--------|----------|------|-------|------------|------------|--------|
| 10101 | CS-101    | 1      | Fall     | 2009 | 10101 | Srinivasan | Comp. Sci. | 65000  |
| 10101 | CS-315    | 1      | Spring   | 2010 | 12121 | Wu         | Finance    | 90000  |
| 10101 | CS-347    | 1      | Fall     | 2009 | 15151 | Mozart     | Music      | 40000  |
| 12121 | FIN-201   | 1      | Spring   | 2010 | 22222 | Einstein   | Physics    | 95000  |
| 15151 | MU-199    | 1      | Spring   | 2010 | 32343 | El Said    | History    | 60000  |
| 22222 | PHY-101   | 1      | Fall     | 2009 | 33456 | Gold       | Physics    | 87000  |
| 32343 | HIS-351   | 1      | Spring   | 2010 | 45565 | Katz       | Comp. Sci. | 75000  |
| 45565 | CS-101    | 1      | Spring   | 2010 | 58583 | Califieri  | History    | 62000  |
| 45565 | CS-319    | 1      | Spring   | 2010 | 76543 | Singh      | Finance    | 80000  |
| 76766 | BIO-101   | 1      | Summer   | 2009 | 76766 | Crick      | Biology    | 72000  |
| 76766 | BIO-301   | 1      | Summer   | 2010 | 83821 | Brandt     | Comp. Sci. | 92000  |
| 83821 | CS-190    | 1      | Spring   | 2009 | 98345 | Kim        | Elec. Eng. | 80000  |

Teaches

Instructor

| course_id | sec_id | semester | year | building | room_number | time_slot_id | course_id | prereq_id |
|-----------|--------|----------|------|----------|-------------|--------------|-----------|-----------|
| BIO-101   | 1      | Summer   | 2009 | Painter  | 514         | B            | BIO-301   | BIO-101   |
| BIO-301   | 1      | Summer   | 2010 | Painter  | 514         | A            | BIO-399   | BIO-101   |
| CS-101    | 1      | Fall     | 2009 | Packard  | 101         | H            | CS-190    | CS-101    |
| CS-101    | 1      | Spring   | 2010 | Packard  | 101         | F            | CS-315    | CS-101    |
| CS-190    | 1      | Spring   | 2009 | Taylor   | 3128        | E            | CS-319    | CS-101    |
| CS-190    | 2      | Spring   | 2009 | Taylor   | 3128        | A            | CS-347    | CS-101    |
| CS-315    | 1      | Spring   | 2010 | Watson   | 120         | D            | EE-181    | PHY-101   |
| CS-319    | 1      | Spring   | 2010 | Watson   | 100         | B            |           |           |
| CS-319    | 2      | Spring   | 2010 | Taylor   | 3128        | C            |           |           |
| CS-347    | 1      | Fall     | 2009 | Taylor   | 3128        | A            |           |           |
| EE-181    | 1      | Spring   | 2009 | Taylor   | 3128        | C            |           |           |
| FIN-201   | 1      | Spring   | 2010 | Packard  | 101         | B            |           |           |
| HIS-351   | 1      | Spring   | 2010 | Painter  | 514         | C            |           |           |
| MU-199    | 1      | Spring   | 2010 | Packard  | 101         | D            |           |           |
| PHY-101   | 1      | Fall     | 2009 | Watson   | 100         | A            |           |           |

Section

Prereq

| dept_name  | building | budget | course_id | title                      | dept_name  | credits |
|------------|----------|--------|-----------|----------------------------|------------|---------|
| Biology    | Watson   | 90000  | BIO-101   | Intro. to Biology          | Biology    | 4       |
| Comp. Sci. | Taylor   | 100000 | BIO-301   | Genetics                   | Biology    | 4       |
| Elec. Eng. | Taylor   | 85000  | BIO-399   | Computational Biology      | Biology    | 3       |
| Finance    | Painter  | 120000 | CS-101    | Intro. to Computer Science | Comp. Sci. | 4       |
| History    | Painter  | 50000  | CS-190    | Game Design                | Comp. Sci. | 4       |
| Music      | Packard  | 80000  | CS-315    | Robotics                   | Comp. Sci. | 3       |
| Physics    | Watson   | 70000  | CS-319    | Image Processing           | Comp. Sci. | 3       |
|            |          |        | CS-347    | Database System Concepts   | Comp. Sci. | 3       |
|            |          |        | EE-181    | Intro. to Digital Systems  | Elec. Eng. | 3       |
|            |          |        | FIN-201   | Investment Banking         | Finance    | 3       |
|            |          |        | HIS-351   | World History              | History    | 3       |
|            |          |        | MU-199    | Music Video Production     | Music      | 3       |
|            |          |        | PHY-101   | Physical Principles        | Physics    | 4       |

Department

Course