Relational Algebra and the SQL Query Language

Paul Fodor

CSE316: Fundamentals of Software Development

Stony Brook University

http://www.cs.stonybrook.edu/~cse316

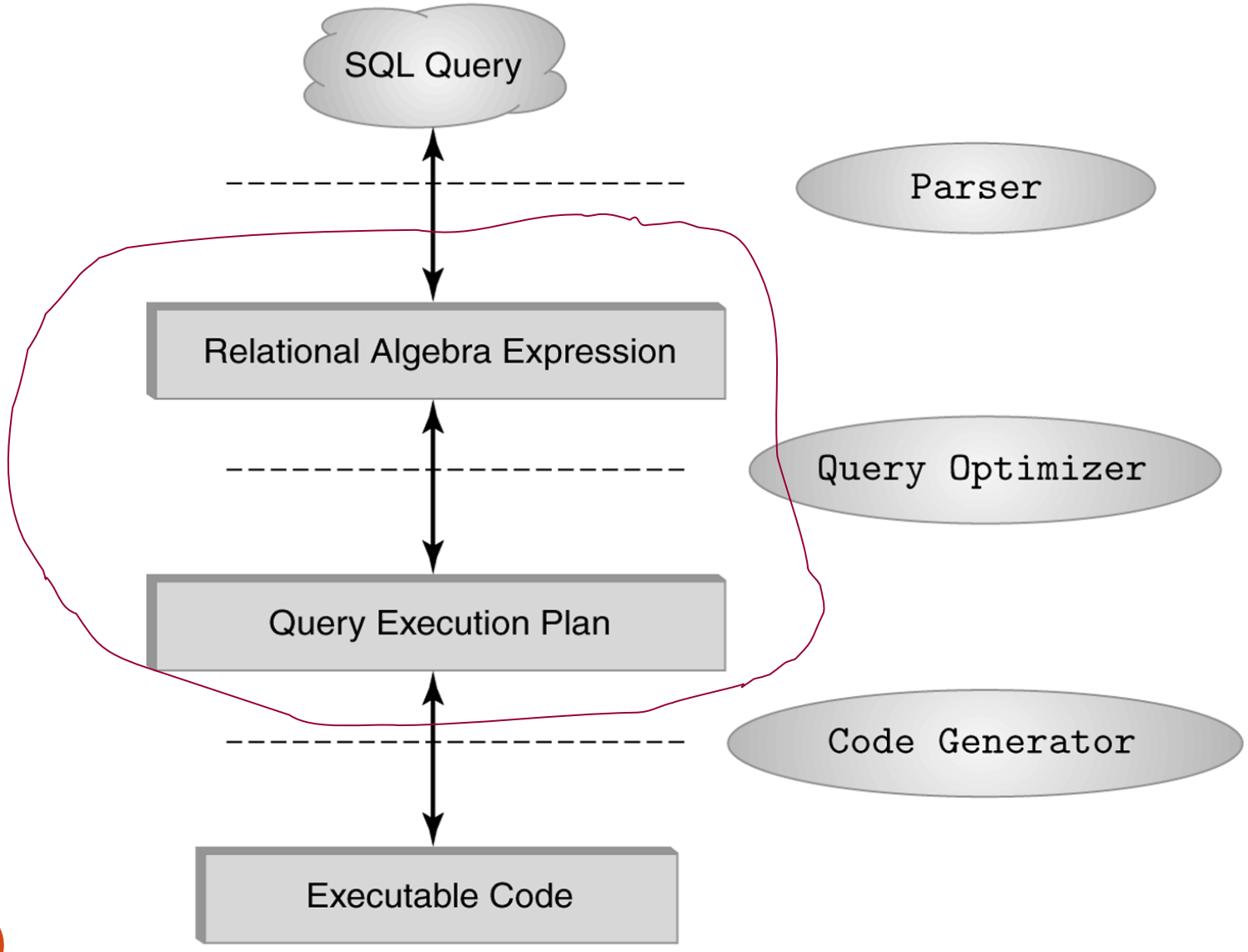
Relational Query Languages

- Now that we know how to create a database, the next step is to learn how to query it to retrieve the information needed for some particular application.
- A *database query language* is a special-purpose programming language designed for retrieving information stored in a database

Relational Query Languages

- Languages for describing queries on a relational databases:
  - *Relational Algebra*
    - Intermediate language used within the DBMS
    - Procedural
      - the **query optimizer** converts the query algebraic expression into an equivalent faster **query execution plan**
  - *Structured Query Language* (SQL)
    - The predominant application-level query language
    - Declarative
      - translated internally by the DBMS into relational algebra expressions (or other calculi, such as tuple relational calculus)

The Role of Relational Algebra in a DBMS



What is an Algebra?

- A language based on operators and a domain of values
- Operators map values taken from the domain into other domain values
  - Hence, an expression involving operators and arguments produces a value in the domain
- When the domain is a set of all relations (and the operators that we will described later), we get the *relational algebra*
- We refer to the expression as a *query* and the value produced as the *query result*

Relational Algebra

- *Domain*: a set of relations
- *Basic operators*:
  - select
  - project
  - union
  - set difference
  - Cartesian product
- *Derived operators*:
  - intersection
  - division
  - join

Relational Algebra

- *Procedural*: Relational expression specifies query by describing an algorithm (the sequence in which operators are applied) for determining the result of an expression.

Select Operator

$$\sigma_{condition}(relation)$$

- Produces a relation/table containing subset of rows of argument table satisfying a condition
- Example:

Person

Id	*Name*	*Address*	*Hobby*
1123	John	123 Main	stamps
1123	John	123 Main	coins
5556	Mary	7 Lake Dr	hiking
9876	Bart	5 Pine St	stamps

$\sigma_{Hobby='stamps'}(\text{Person})$

Id	*Name*	*Address*	*Hobby*
1123	John	123 Main	stamps
9876	Bart	5 Pine St	stamps

Selection Condition

- Simple selection condition:
  - <*attribute*> *operator* <*constant/attribute*>

Operator	Description
=	Equal
>	Greater Than
<	Less Than
>=	Greater or Equal
<=	Less or Equal
<>	Not Equal

Operator	Description
In	Specify Multiple Possible Values for Column
BETWEEN	Within a range
LIKE	Pattern Search

- And Boolean expressions:
  - <*condition*> AND <*condition*>
  - <*condition*> OR <*condition*>
  - NOT <*condition*>

Selection Condition - Examples

- $\sigma_{Id>3000 \text{ OR } Hobby='hiking'}(\text{Person})$
- $\sigma_{Id>3000 \text{ AND } Id<3999}(\text{Person})$
- $\sigma_{\text{NOT}(Hobby='hiking')}(\text{Person})$
- $\sigma_{Hobby<>'hiking'}(\text{Person})$

Project Operator

$$\pi_{attribute\ list}(relation)$$

- Produces the relation/table containing subset of columns of the argument table
- Example:

Person

Id	*Name*	*Address*	*Hobby*
1123	John	123 Main	stamps
1123	John	123 Main	coins
5556	Mary	7 Lake Dr	hiking
9876	Bart	5 Pine St	stamps

$\pi_{Name,Hobby}(\text{Person})$

Name	*Hobby*
John	stamps
John	coins
Mary	hiking
Bart	stamps

Project Operator

- Relational Algebra has a set semantics: No Duplicates!

Person

Id	*Name*	*Address*	*Hobby*
1123	John	123 Main	stamps
1123	John	123 Main	coins
5556	Mary	7 Lake Dr	hiking
9876	Bart	5 Pine St	stamps

$\pi_{Name,Address}$(Person)

Name	*Address*
John	123 Main
Mary	7 Lake Dr
Bart	5 Pine St

The result is a relation/table (no duplicates by definition), so the result can have fewer tuples than the original!

Relational Algebra Expressions

$$\pi_{Id,\ Name}\ (\sigma_{Hobby='stamps'\ \text{OR}\ Hobby='coins'}\ (\text{Person}))$$

Id	*Name*	*Address*	*Hobby*
1123	John	123 Main	stamps
1123	John	123 Main	coins
5556	Mary	7 Lake Dr	hiking
9876	Bart	5 Pine St	stamps

Person

Id	*Name*
1123	John
9876	Bart

Result

Set Operators

- A Relation is a **set** of tuples, so set operations should apply: $\cap$, $\cup$, $-$ (set difference)
- The result of combining two relations with a set operator is also a relation => all its elements must be tuples having the same structure
- Hence, the scope of set operations is limited to ***union compatible relations***

Union Compatible Relations

- Two relations are *union compatible* if:
  - Both have same number of columns
  - Names of attributes are the same in both
  - Attributes with the same name in both relations have the same domain
- Union compatible relations can be combined using *union*, *intersection*, and *set difference*

Union Example

Tables:

Person (*SSN*, *Name*, *Address*, *Hobby*)

Professor (*Id*, *Name*, *Office*, *Phone*)

are <u>not</u> union compatible.

But

$$\pi_{Name}(\text{Person}) \text{ and } \pi_{Name}(\text{Professor})$$

<u>are</u> union compatible so

$$\pi_{Name}(\text{Person}) - \pi_{Name}(\text{Professor})$$

makes sense.

Cartesian Product

- If R and S are two relations, $R \times S$ is the set of all concatenated tuples $<x,y>$, where x is a tuple in R and y is a tuple in S
- $R \times S$ is <u>expensive to compute</u>:
  - Quadratic in the number of rows

A	*B*
x1	x2
x3	x4

R

C	*D*
y1	y2
y3	y4

S

A	*B*	*C*	*D*
x1	x2	y1	y2
x1	x2	y3	y4
x3	x4	y1	y2
x3	x4	y3	y4

$R \times S$

Renaming

- The result of expression evaluation is a relation
- The attributes of relation must have distinct names. This is not guaranteed with Cartesian product
  - e.g., suppose in previous example *a* and *c* have the same name
- *Renaming operator* tidies this up. To assign the names A_1, A_2,… A_n to the attributes of the *n* column relation produced by expression *expr* use

$$expr\ [A_1, A_2, \ldots A_n]$$

Renaming Example

Transcript (*StudId*, *CrsCode*, *Semester*, *Grade*)

Teaching (*ProfId*, *CrsCode*, *Semester*)

$$\pi_{StudId,\ CrsCode}(\text{Transcript})[StudId,\ CrsCode1]$$

$$\times\ \pi_{ProfId,\ CrsCode}(\text{Teaching})\ [ProfId,\ CrsCode2]$$

This is a relation with 4 attributes:

StudId, *CrsCode1*, *ProfId*, *CrsCode2*

Derived Operation: Join

A (*general* or *theta*) *join* of R and S is the expression

$$R \bowtie_{\textit{join-condition}} S$$

where *join-condition* is a *conjunction* of terms:

$$A_i \ \textit{operator}\ B_i$$

in which A_i is an attribute of R; B_i is an attribute of S; and *operator* is one of =, <, >, ≥ ≠, ≤.

The meaning is:

$$\sigma_{\textit{join-condition}'}(R \times S)$$

where *join-condition* and *join-condition*′ are the same, except for possible renamings of attributes (next)

Join and Renaming

- **Problem**: *R* and *S* might have attributes with the same name – in which case the Cartesian product is not defined
- **Solutions**:
  1. Rename attributes prior to forming the product and use new names in *join-condition*. OR
  2. Qualify common attribute names with relation names (thereby disambiguating the names). For instance: Transcript.*CrsCode* or Teaching.*CrsCode*
     - This solution is nice, but doesn't always work: consider

$$R \bowtie_{join\_condition} R$$

how do we know which R is meant?

Theta Join – Example

Employee(*Name*,*Id*,*MngrId*,*Salary*)
Manager(*Name*,*Id*,*Salary*)

Output the names of all employees that earn more than their managers.

$$\pi_{\text{Employee}.Name}(\text{Employee} \bowtie_{MngrId=Id \text{ AND } Salary>Salary} \text{Manager})$$

The inner join yields a table with attributes:

Employee.*Name*, Employee.*Id*, Employee.*Salary*, *MngrId*
Manager.*Name*, Manager.*Id*, Manager.*Salary*

Then we project only on the Employee.*Name*.

Equijoin Join - Example

Equijoin: Join condition is a conjunction of *equalities*.

$$\pi_{Name,CrsCode}(\text{Student} \bowtie_{Id=StudId} \sigma_{Grade='A'}(\text{Transcript}))$$

Student

Id	*Name*	*Addr*	*Status*
111	John	…..	…..
222	Mary	…..	…..
333	Bill	…..	…..
444	Joe	…..	…..

Transcript

StudId	*CrsCode*	*Sem*	*Grade*
111	CSE305	S00	B
222	CSE306	S99	A
333	CSE304	F99	A

Mary	CSE306
Bill	CSE304

The equijoin is used very frequently since it combines related data in different relations on foreign keys.

Natural Join

- Special case of equijoin:
  - join condition equates *all* and *only* those attributes with the same name (condition doesn't have to be explicitly stated)
  - duplicate columns eliminated from the result (i.e., only 1 copy is kept)

Transcript (*StudId*, *CrsCode*, *Sem*, *Grade*)
Teaching (*ProfId*, *CrsCode*, *Sem*)

$$\text{Transcript} \bowtie \text{Teaching} =$$

$$\pi_{StudId,\ Transcript.CrsCode,\ Transcript.Sem,\ Grade,\ ProfId}$$

$$(\ \text{Transcript} \bowtie_{CrsCode=CrsCode\ \text{AND}\ Sem=Sem} \text{Teaching}\ )$$

$$[StudId,\ CrsCode,\ Sem,\ Grade,\ ProfId\ ]$$

Natural Join

- More generally:

$$R \bowtie S = \pi_{attr\text{-}list} (\sigma_{join\text{-}cond} (R \times S))$$

where

$$attr\text{-}list = attributes(R) \cup attributes(S)$$

(duplicates are eliminated) and *join-cond* has the form:

$$R.A_1 = S.A_1 \text{ AND } \ldots \text{ AND } R.A_n = S.A_n$$

where

$$\{A_1 \ldots A_n\} = attributes(R) \cap attributes(S)$$

Natural Join Example

- List all Ids of students who took at least two different courses:

$$\pi_{StudId}\,(\,\sigma_{CrsCode \neq CrsCode2}\,(\,\text{Transcript} \bowtie \text{Transcript}\,[StudId, CrsCode2, Sem2, Grade2]\,))$$

We don't want to join on *CrsCode*, *Sem*, and *Grade* attributes, hence renaming!

Division (/,÷)

- Goal: Produce the tuples in one relation, r, that match *all* tuples in another relation, s
  - $r\ (A_1, \ldots A_n, B_1, \ldots B_m)$
  - $s\ (B_1\ \ldots B_m)$
  - r/s, with attributes $A_1, \ldots A_n$, is the set of all tuples <*a*> such that for every tuple <*b*> in *s*, <*a,b*> is in *r*
- Can be expressed in terms of projection, set difference, and cross-product:

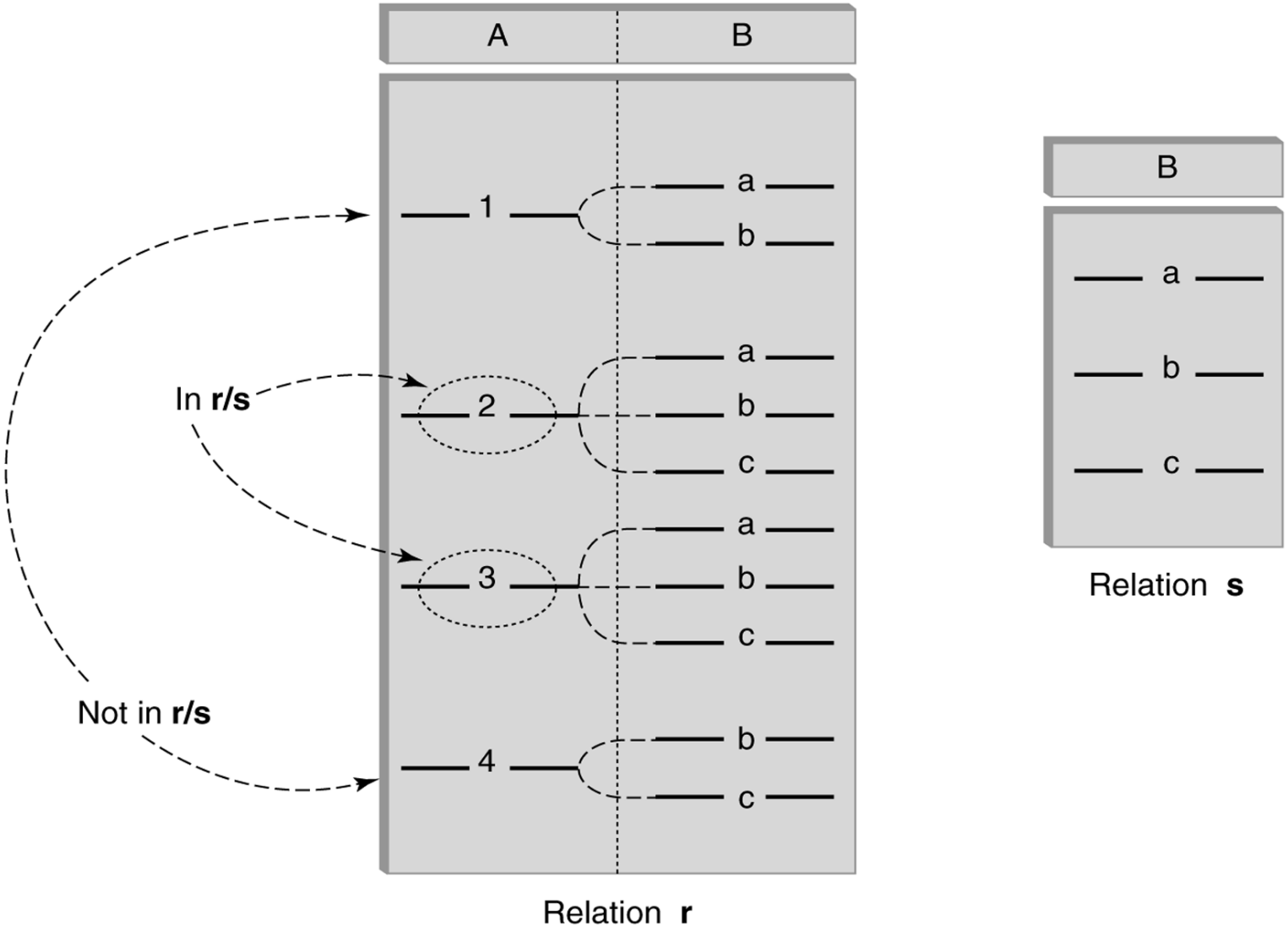
$$\text{let } t := \pi_{A1,\ldots,An}(r) \times s \qquad \text{has the same attributes with } r$$

$$\text{let } u := t - r$$

$$\text{let } v := \pi_{A1,\ldots,An}(u)$$

$$r/s = \pi_{A1,\ldots,An}(r) - v$$

Division (/,÷)



Division Example

- List the Ids of students who have passed *all* courses that were taught in Fall 2020
  - *Numerator*:
    - *StudId* and *CrsCode* for every course passed by every student:

$$\pi_{StudId,\ CrsCode}\left(\sigma_{Grade \neq 'F'}\ (\text{Transcript})\right)$$

  - *Denominator*:
    - *CrsCode* of all courses taught in Fall 2020

$$\pi_{CrsCode}\left(\sigma_{Semester='F2020'}(\text{Teaching})\right)$$

  - Result is *Numerator / Denominator*

The Query Sublanguage of SQL

- SQL – Structured Query Language
  - Syntax:

SELECT <column_list> FROM <table_name_list>

[WHERE condition]

[GROUP BY <column_name>]

[HAVING <condition>]

[ORDER BY <column_name> [ASC|DESC]]

Remember our Schema for the Student Registration System

Student (*Id*, *Name*, *Addr*, *Status*)
Professor (*Id*, *Name*, *DeptId*)
Course (*DeptId*, *CrsCode*, *CrsName*, *Descr*)
Transcript (*StudId, CrsCode, Semester*, *Grade*)
Teaching (*ProfId*, *CrsCode, Semester*)
Department (*DeptId*, *Name*)

The Query Sublanguage of SQL

```
SELECT  C.CrsName AS Course
FROM  Course C
WHERE  C.DeptId = 'CSE'
```

- Evaluation strategy:
  - FROM clause produces Cartesian product of listed tables
    - C is a *tuple variable (alias* for the relation*)* and it ranges over rows of Course.
  - WHERE filters only rows satisfying the condition
  - SELECT clause retains/projects only the listed columns
- Equivalent to: $\pi_{CrsName}\,\sigma_{DeptId='CSE'}(\text{Course})$

Join Queries

- List the names of all the courses taught in F2020

```
SELECT  C.CrsName
FROM  Course C, Teaching T
WHERE  C.CrsCode=T.CrsCode AND T.Semester='F2020'
```

- Join condition "C.*CrsCode*=T.*CrsCode*"
  - relates facts to each other
- Selection condition "T.*Semester*='F2020'
  - eliminates irrelevant rows
- Followed by projection on CrsName
  - Note SELECT "*" means all the attributes

Correspondence Between SQL and Relational Algebra

```
SELECT   C.CrsName
FROM     Course C, Teaching T
WHERE    C.CrsCode = T.CrsCode  AND  T.Semester = 'F2020'
```

Equivalent relational algebra expressions:

$$\pi_{CrsName}(\text{Course} \bowtie \sigma_{Semester='F2020'}(\text{Teaching}))$$

$$\pi_{CrsName}(\sigma_{Sem='F2020'}(\text{Course} \bowtie \text{Teaching}))$$

$$\pi_{CrsName}\ \sigma_{C\_CrsCode=T\_CrsCode\ AND\ Semester='F2020'}$$
$$(\text{Course}\ [C\_CrsCode, DeptId, CrsName, Desc]$$
$$\times\ \text{Teaching}\ [ProfId, T\_CrsCode, Semester])$$

- Relational algebra expressions are procedural
  - Which of the equivalent expressions is most efficient? First one.

Self-join Queries

Find the Ids of all professors who taught at least two courses in the same semester:

```
SELECT  T1.ProfId
FROM  Teaching T1, Teaching T2
WHERE  T1.ProfId = T2.ProfId
    AND  T1.Semester = T2.Semester
    AND  T1.CrsCode <> T2.CrsCode
```

Tuple variables are essential in this query!

Equivalent to:

$$\pi_{ProfId}(\sigma_{T1.CrsCode \neq T2.CrsCode}(\text{Teaching}[ProfId, T1.CrsCode, Semester] \bowtie \text{Teaching}[ProfId, T2.CrsCode, Semester]))$$

Duplicates

- Duplicate rows not allowed in a relation
- However, duplicate elimination from query result is costly and not done by default; must be explicitly requested:

```
SELECT DISTINCT .....
FROM .....
```

Expressions

Equality and comparison operators apply to strings (based on lexical ordering)

WHERE S.*Name* < 'P'

Concatenate operator || applies to strings

WHERE S.*Name* || '--' || S.*Address* = ….

Expressions can also be used in SELECT clause:

SELECT S.*Name* || '--' || S.*Address* AS *NmAdd*
FROM Student S

Set Operators

- SQL provides UNION, EXCEPT (set difference), and INTERSECT for union compatible tables
  - Example: Find all professors in the CSE Department and all professors that have taught CSE courses

```
(SELECT   P.Name
 FROM   Professor P, Teaching T
 WHERE  P.Id=T.ProfId AND T.CrsCode LIKE 'CSE%')
UNION
(SELECT  P.Name
 FROM  Professor P
 WHERE  P.DeptId = 'CSE')
```

Nested Queries

- List all courses that were not taught in F2020:

```
SELECT C.CrsName
FROM Course C
WHERE C.CrsCode  NOT  IN
      (SELECT T.CrsCode      --subquery
       FROM Teaching T
       WHERE T.Sem = 'F2020')
```

Evaluation strategy: subquery evaluated once to produces set of courses taught in F2020. Each row (as C) tested against this set.

Correlated Nested Queries

Output a row <*prof*, *dept*> if *prof* has taught a course in *dept*.

```
SELECT  P.Name, D.Name                    --outer query
  FROM Professor P, Department D
  WHERE  P.Id  IN
            -- set of  all ProfId's who have taught a course in D.DeptId
         (SELECT T.ProfId                 --subquery
          FROM Teaching T, Course C
          WHERE T.CrsCode=C.CrsCode  AND
                C.DeptId=D.DeptId         --correlation
         )
```

- Tuple variables T and C are *local* to subquery, while P and D are *global* to subquery
- *Correlation*: subquery uses a global variable D
- The value of D.*DeptId* parameterizes an evaluation of the subquery
- Subquery must be re-evaluated for each distinct value of D.*DeptId*
- ***Correlated queries can be expensive to evaluate!!!***

Division in SQL

- *Query type*: Find the subset of items in one set that are related to *all* items in another set
- *Example*:

Find professors who taught courses in *all* departments

Contains row <*p*,*d*> if professor *p* taught a course in department *d*

ProfId	*DeptId*

DeptId

All department Ids

$$\pi_{\text{ProfId, DeptId}}(\text{Teaching} \bowtie \text{Course}) \ / \ \pi_{\text{DeptId}}(\text{Department})$$

Division in SQL

- *Strategy for implementing division in SQL*:
  - Find set, A, of all departments in which a particular professor, *p*, has taught a course
  - Find set, B, of all departments
  - Output *p* if A $\supseteq$ B, or, equivalently, if B–A is empty

Division in SQL

```
SELECT  P.Id
FROM  Professor P
WHERE
   NOT EXISTS
   (SELECT  D.DeptId              -- set B of all dept Ids
    FROM  Department D
       EXCEPT
    SELECT  C.DeptId              -- set A of dept Ids of depts in
                                  -- which P taught a course
    FROM  Teaching T, Course C
    WHERE  T.ProfId=P.Id      -- global variable
           AND  T.CrsCode=C.CrsCode)
```

Aggregates

- Functions that operate on sets:
  - COUNT, SUM, AVG, MAX, MIN
- Produce numbers (not tables)

```
SELECT COUNT(*)
FROM  Professor P
```

```
SELECT MAX (Salary)
FROM  Employee E
```

Aggregates

Count the number of courses taught in F2020:

```
SELECT COUNT (T.CrsCode)
FROM Teaching T
WHERE  T.Semester = 'F2020'
```

But if multiple sections of same course are taught, use:

```
SELECT COUNT (DISTINCT T.CrsCode)
FROM Teaching T
WHERE  T.Semester = 'F2020'
```

Grouping

- But how do we compute the number of courses taught in F2020 *per professor*?
  - Strategy 1: Fire off a separate query for <u>each</u> professor:

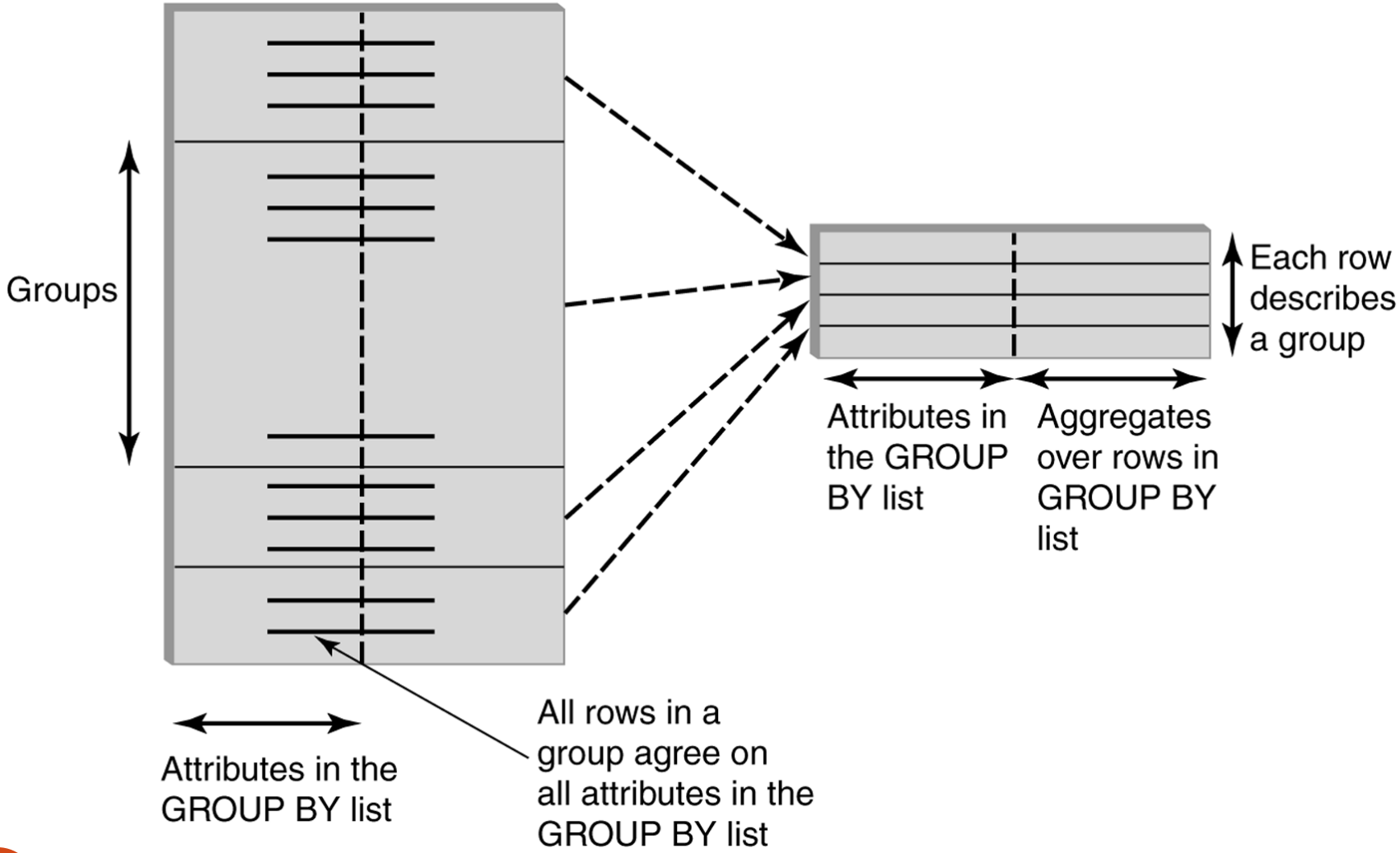
```
SELECT   COUNT(T.CrsCode)
FROM     Teaching T
WHERE    T.Semester = 'F2020' AND T.ProfId = 123456789
```

    - Cumbersome: we have to query prof. by prof.
  - Strategy 2: define a special *grouping operator*:

```
SELECT     T.ProfId, COUNT(DISTINCT T.CrsCode)
FROM       Teaching T
WHERE      T.Semester = 'F2020'
GROUP BY   T.ProfId
```

In the SELECT part of a SQL query that contains GROUP BY, you can only include <u>attributes of GROUP BY list and aggregates on attributes not in that list</u>

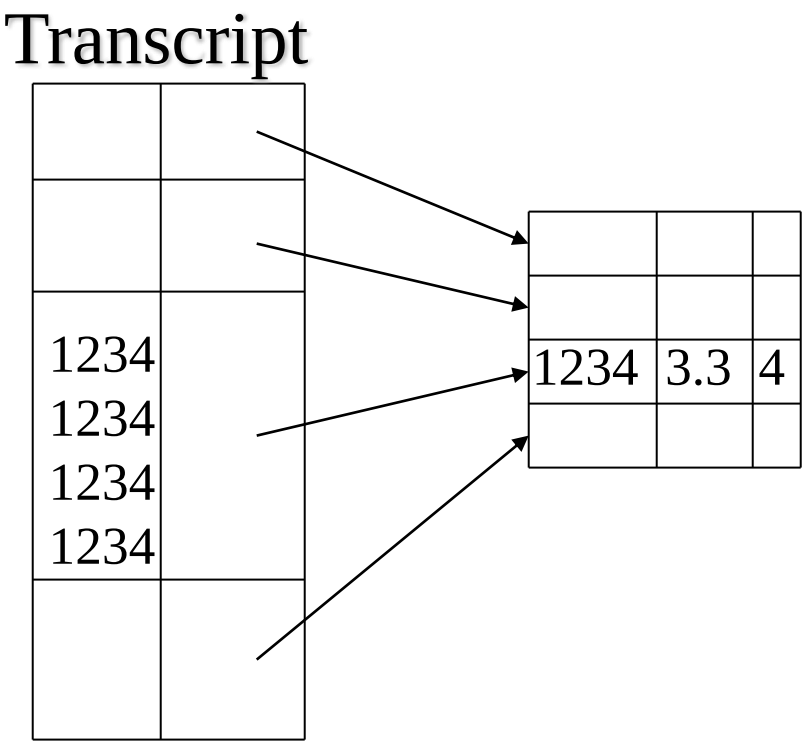
GROUP BY



GROUP BY – multiple aggregates in one query

Find the: student's *Id*, avg grade and number of courses

```
SELECT T.StudId, AVG(T.Grade), COUNT (*)
FROM Transcript T
GROUP BY T.StudId
```



HAVING Clause

- Eliminate unwanted groups (analogous to WHERE clause, but works on groups instead of individual tuples)
- HAVING condition is constructed from attributes of GROUP BY list and aggregates on attributes not in that list
- Filter the previous example for students with GPA > 3.5

Find the: student's *Id*, avg grade and number of courses

```
SELECT  T.StudId,
        AVG(T.Grade)  AS  CumGpa,
        COUNT (*)  AS  NumCrs
FROM   Transcript  T
WHERE   T.CrsCode  LIKE  'CS%'
GROUP BY  T.StudId
HAVING  AVG (T.Grade) > 3.5
```

Example

- Output the name and address of all seniors on the Dean's List

```
SELECT  S.Id, S.Name
FROM    Student S, Transcript T
WHERE   S.Id = T.StudId  AND  S.Status = 'senior'

GROUP BY  S.Id              -- wrong
          S.Id, S.Name      -- right

HAVING AVG (T.Grade) > 3.5  AND  SUM (T.Credit) > 90
```

Every attribute that occurs in SELECT *clause must also occur in* GROUP BY *or it must be an aggregate. S.Name does not.*

Aggregates: Proper and Improper Usage

SELECT COUNT (T.*CrsCode*), T. *ProfId*

– *makes no sense (in the absence of* GROUP BY *clause)*

SELECT COUNT (*), AVG (T.*Grade*)

– *but this is OK since it is for the whole relation*

SELECT … FROM …
WHERE T.*Grade* > COUNT (SELECT ….)

– *aggregate cannot be applied to the result of a* SELECT *statement*

***Note: never use an aggregate in* WHERE**

ORDER BY Clause

- Causes rows to be output in a specified order

```
SELECT  T.StudId, COUNT (*) AS NumCrs,
        AVG(T.Grade) AS CumGpa
FROM    Transcript T
WHERE   T.CrsCode LIKE 'CS%'
GROUP BY  T.StudId
HAVING  AVG (T.Grade) > 3.5
ORDER BY  DESC  CumGpa,  ASC StudId
```

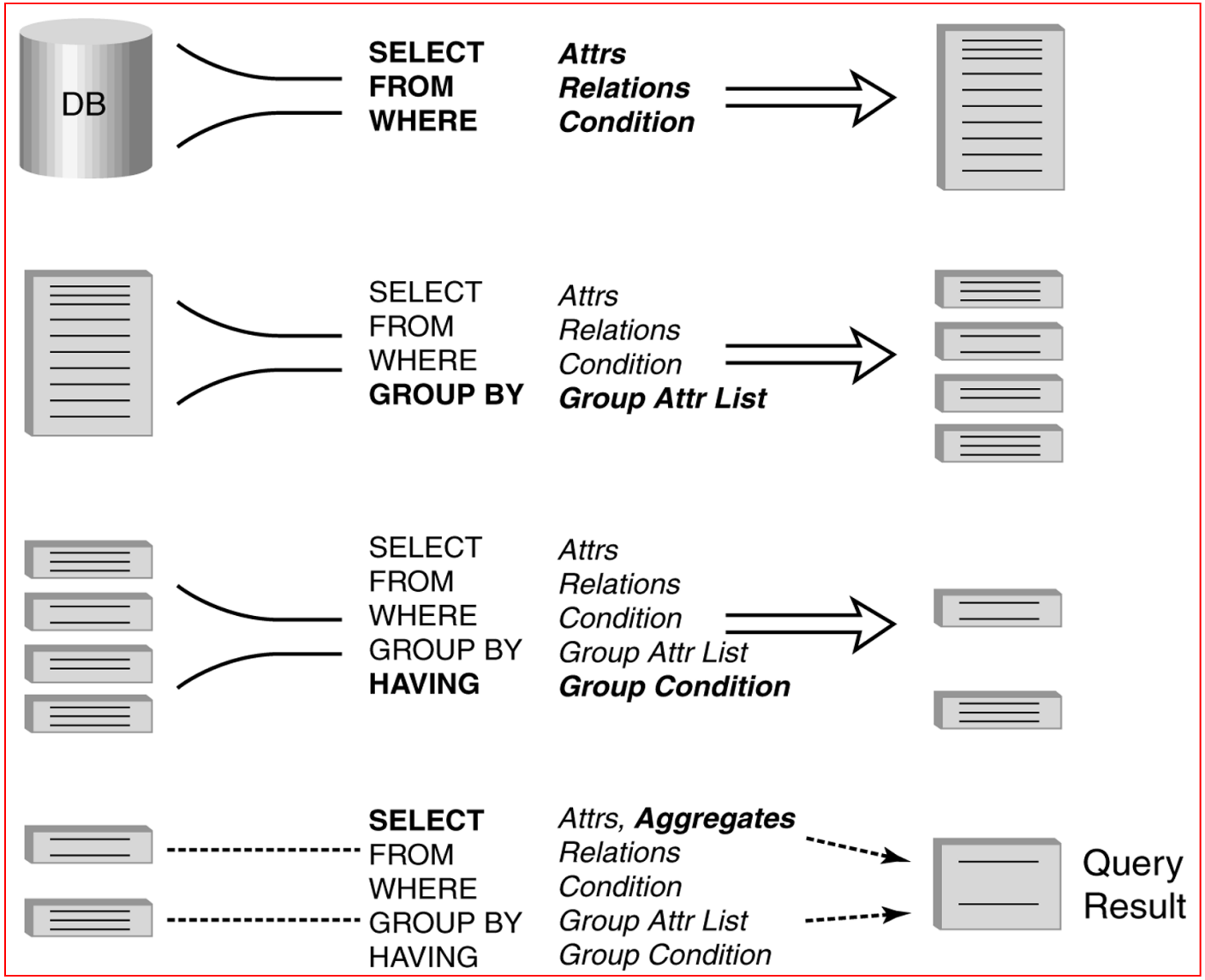
Descending

Ascending

Query Evaluation with GROUP BY, HAVING, ORDER BY

1. Evaluate FROM: produces Cartesian product, A, of tables in FROM list
2. Evaluate WHERE: produces table, B, consisting of rows of A that satisfy WHERE condition
3. Evaluate GROUP BY: partitions B into groups that agree on attribute values in GROUP BY list
4. Evaluate HAVING: eliminates groups in B that do not satisfy HAVING condition
5. Evaluate SELECT: produces table C containing a row for each group. Attributes in SELECT list limited to those in GROUP BY list and aggregates over group
6. Evaluate ORDER BY: orders rows of C

Order of Operations with GroupBy&Having



Views

- Used as a relation, but rows are not physically stored.
  - The contents of a view is *computed* when it is used within an SQL statement
- View is the result of a SELECT statement over other views and base relations
- When used in an SQL statement, the view definition is substituted for the view name in the statement
  - As SELECT statement nested in FROM clause

View Example

```
CREATE VIEW CumGpa (StudId, Cum) AS
  SELECT T.StudId, AVG (T.Grade)
  FROM Transcript T
  GROUP BY T.StudId
```

- Used in other SELECT statements:

```
SELECT S.Name, C.Cum
FROM CumGpa C, Student S
WHERE C.StudId = S.StudId AND C.Cum > 3.5
```

View Benefits

- *Access Control*: Users not granted access to base tables. Instead they are granted access to the view of the database appropriate to their needs.
  - *External schema* is composed of views.
  - View allows owner to provide SELECT access to a subset of columns (analogous to providing UPDATE and INSERT access to a subset of columns)

Views – Limiting Visibility

Grade projected out

```
CREATE VIEW PartOfTranscript (StudId, CrsCode, Semester) AS
    SELECT T. StudId, T.CrsCode, T.Semester        -- limit columns
    FROM Transcript T
    WHERE T.Semester = 'F2020'                     -- limit rows
```

Give permissions to access data through view:

```
GRANT SELECT ON PartOfTranscript TO joe
```

This would have been analogous to:

```
GRANT SELECT (StudId,CrsCode,Semester)
                    ON Transcript TO joe
```

on regular tables, <u>if</u> SQL allowed attribute lists in GRANT SELECT

View Benefits

- *Customization*: Users need not see full complexity of database. View creates the illusion of a simpler database customized to the needs of a particular category of users
- A view is *similar in many ways to a subroutine* in standard programming
  - Can be reused in multiple queries

Nulls and *undefined*

- *Arithmetic expression*: x *op y* (where *op* is +, –, *, etc.) has value NULL if x or y is NULL
  - WHERE (T. *price*/T.*cost*) > 2 -- *NULL*/T.*cost is NULL*

- *Conditions*: *x op y* (where *op* is <, >, <>, =, etc.) has value *undefined*/*unknown* (*U*) <u>when either x or y is null</u>
  - WHERE T.*cost* > T.*price*

Nulls and *undefined*

- WHERE clause uses a *three-valued logic – T, F, U(ndefined)* – to filter rows. Portion of truth table:

C1	*C2*	*C1* AND *C2*	*C1* OR *C2*
T	U	U	T
F	U	F	U
U	U	U	U

- **<u>Rows are discarded if WHERE condition is</u>** *F(alse)* or **<u>U(*ndefined*)</u>**

Nulls

- *Aggregates*: COUNT counts NULLs like any other value; other aggregates ignore NULLs

```
SELECT  COUNT (T.CrsCode),  AVG (T.Grade)
FROM    Transcript T
WHERE  T.StudId = '1234'
```

SQL INNER JOIN Keyword

- INNER JOIN/JOIN keywords selects all rows from both tables as long as there is a match between the columns in both tables.

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name=table2.column_name;
```

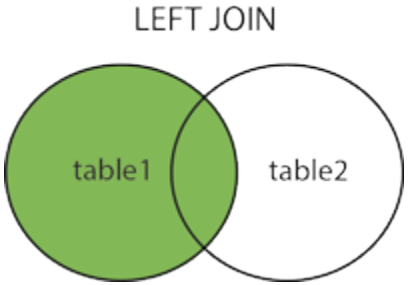
- or:

```
SELECT column_name(s)
FROM table1
JOIN table2
ON table1.column_name=table2.column_name;
```

- INNER JOIN is the same as JOIN

SQL LEFT JOIN Keyword

- The LEFT JOIN keyword returns all rows from the left table (table1), with the matching rows in the right table (table2).
- The result is NULL in the right side when there is no match.



SQL LEFT JOIN Keyword

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name=table2.column_name;
```

- or:

```
SELECT column_name(s)
FROM table1
LEFT OUTER JOIN table2
ON table1.column_name=table2.column_name;
```

- LEFT JOIN is the same as LEFT OUTER JOIN

SQL RIGHT JOIN Keyword

- The RIGHT JOIN keyword returns all rows from the right table (table2), with the matching rows in the left table (table1).
  - The result is NULL in the left side when there is no match.

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name=table2.column_name;
```

- or:

```
SELECT column_name(s)
FROM table1
RIGHT OUTER JOIN table2
ON table1.column_name=table2.column_name;
```

SQL FULL OUTER JOIN

- SQL FULL OUTER JOIN Keyword: combines the result of both LEFT and RIGHT joins.

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name=table2.column_name;
```

Comma join vs. JOIN

```
SELECT pers.FirstName, pers.LastName, p.Name, p.ListPrice
FROM Sales.Customer AS c, Sales.SalesOrderHeader AS soh,
   Sales.SalesOrderDetail AS sod, Production.Product AS p,
   person.Person AS pers
WHERE p.ProductID = sod.ProductID  AND c.CustomerID = soh.CustomerID
   AND soh.SalesOrderID = sod.SalesOrderID AND c.PersonID = pers.BusinessEntityID;
```

- VS

```
SELECT pers.FirstName, pers.LastName, p.Name, p.ListPrice
FROM Sales.Customer AS c
INNER JOIN Sales.SalesOrderHeader AS soh ON c.CustomerID = soh.CustomerID
INNER JOIN Sales.SalesOrderDetail AS sod ON sod.SalesOrderID = soh.SalesOrderID
INNER JOIN Production.Product AS p ON sod.ProductID = p.ProductID
INNER JOIN person.Person AS pers ON pers.BusinessEntityID = c.PersonID;
```

JOIN as comma join

- LEFT OUTER JOINs in the comma form: by changing the equal sign to an asterisk + equal sign (*=) or a RIGHT OUTER JOIN by changing the equal sign to an equal sign + asterisk (=*).
- The JOIN syntax makes it easy to add or eliminate the tables for trouble-shooting by commenting out the line.

SQL LIKE Operator

- The LIKE operator is used to search for a specified pattern in a column.

```
SELECT column_name(s)
FROM table_name
WHERE column_name LIKE pattern;
```

- selects all customers with a City starting with the letter "s" AND a Country containing the pattern "land" AND the Country NOT LIKE '%green%':

```
SELECT * FROM Customers
WHERE City LIKE 's%'
AND Country LIKE '%land%'
AND Country NOT LIKE '%green%';
```

SQL Wildcard Characters

- A wildcard character can be used to substitute for any other character(s) in a string.

Wildcard	Description
%	A substitute for zero or more characters
_	A substitute for a single character
[*charlist*]	Sets and ranges of characters to match
[^*charlist*] or [!*charlist*]	Matches only a character NOT specified within the brackets

```
SELECT * FROM Customers
WHERE City LIKE 'L_n_on';
```

SQL BETWEEN Operator

- The BETWEEN operator is used to select values within a range.

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20;
```

SQL IN Operator

- The IN operator allows you to specify multiple values in a WHERE clause.

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1,value2,...);
```

```
SELECT * FROM Customers
WHERE City IN ('Paris','London');
```

MySQL Date Functions

Function	Description
NOW()	Returns the current date and time
CURDATE()	Returns the current date
CURTIME()	Returns the current time
DATE()	Extracts the date part of a date or date/time expression
EXTRACT()	Returns a single part of a date/time
DATE_ADD()	Adds a specified time interval to a date
DATE_SUB()	Subtracts a specified time interval from a date
DATEDIFF()	Returns the number of days between two dates
DATE_FORMAT()	Displays date/time data in different formats

LIMIT

- Limit the number of rows returned
  - MySQL:

```
SELECT column_name(s)
    FROM table_name
    WHERE condition
    LIMIT number;
```

  - SQL Server:

```
SELECT  TOP number|percent column_name(s)
    FROM table_name
    WHERE condition;
```

    - can be a count or a percentage

Modifying Tables – Insert

- Inserting a single row into a table
  - Attribute list can be omitted if it is the same as in CREATE TABLE (but do not omit it)
  - NULL and DEFAULT values can be specified

```
INSERT INTO  Transcript(StudId, CrsCode, Semester, Grade)
VALUES (12345, 'CSE305', 'F2020',  NULL)
```

Bulk Insertion

- Insert the rows output by a SELECT

```
CREATE TABLE DeansList (
        StudId      INTEGER,
        Credits     INTEGER,
        CumGpa      FLOAT,
        PRIMARY KEY StudId )
```

```
INSERT INTO  DeansList (StudId, Credits, CumGpa)
SELECT       T.StudId, 3 * COUNT (*),  AVG(T.Grade)
FROM         Transcript T
GROUP BY     T.StudId
HAVING  AVG (T.Grade) > 3.5  AND  COUNT(*) > 30
```

Modifying Tables – Delete

- Similar to SELECT except:
  - No project list in DELETE clause
  - No Cartesian product in FROM clause (only 1 table name)
  - Rows satisfying WHERE clause (general form, including subqueries, allowed) are deleted instead of output

```
DELETE FROM Transcript T
WHERE  T.Grade IS NULL  AND  T.Semester <> 'F2020'
```

Modifying Data - Update

```
UPDATE  Employee E
SET     E.Salary = E.Salary * 1.05
WHERE   E.Department = 'R&D'
```

- Updates rows in a single table
- All rows satisfying WHERE clause (general form, including subqueries, allowed) are updated

Updating Views

- Question: Since views look like tables to users, can they be updated?
- Answer: Yes – a view update changes the underlying base table to produce the requested change to the view

```
CREATE VIEW   CsReg (StudId, CrsCode, Semester) AS
SELECT        T.StudId, T. CrsCode, T.Semester
FROM          Transcript T
WHERE     T.CrsCode LIKE 'CS%'  AND  T.Semester='F2020'
```

Updating Views - Problem 1

INSERT INTO CsReg (*StudId*, *CrsCode*, *Semester*)
VALUES (1111, 'CSE305', 'F2020')

- **Question**: What value should be placed in attributes of underlying table that have been projected out (e.g., *Grade*)?
- **Answer**: NULL (assuming null allowed in the missing attribute) or DEFAULT

Updating Views - Problem 2

INSERT INTO CsReg (*StudId*, *CrsCode*, *Semester*)
VALUES (1111, 'ECO105', 'F2020')

- **Problem**: New tuple not in view
- **Solution**: Allow insertion (assuming the WITH CHECK OPTION clause has not been appended to the CREATE VIEW statement)

Updating Views - Problem 3

- Update to a view might *not* *uniquely* specify the change to the base table(s) that results in the desired modification of the view (ambiguity)

```
CREATE VIEW   ProfDept (PrName, DeName)  AS
SELECT    P.Name, D.Name
FROM      Professor P, Department D
WHERE     P.DeptId = D.DeptId
```

Updating Views - Problem 3

- Tuple <Smith, CS> can be deleted from ProfDept by:
  - Deleting row for Smith from Professor (but this is inappropriate if he is still at the University)
  - Deleting row for CS from Department (not what is intended)
  - Updating row for Smith in Professor by setting *DeptId* to null (seems like a good idea, but how would the computer know?)

Updating Views - Restrictions

- Updatable views are restricted to those in which
  - No Cartesian product in FROM clause
  - no aggregates, GROUP BY, HAVING

For example, if we allowed:

```
CREATE VIEW AvgSalary (DeptId, Avg_Sal ) AS
    SELECT E.DeptId, AVG(E.Salary)
    FROM Employee E
    GROUP BY E.DeptId
```

then how do we handle:

```
UPDATE AvgSalary
    SET Avg_Sal = 1.1 * Avg_Sal
```

Security Concerns

- Consider a Customer database
  - Allows a user to query for their customer information by entering their SSN:

    SELECT * FROM customers WHERE SSN = $input

    - Expected input would be a 9 digit number (i.e. 123456789)
  - What if user enters: **99 or 1=1**
  - Now, the query will be:

SELECT * FROM customers WHERE SSN = 99 or 1=1

    - This will cause the DB software to return all records in the database!
- Solution: Scan the input data
  - Assure it meets the requirements and format (9 digit number, no special characters)
  - Return an error if it does not
  - Optional: modify the input to remove special characters, keywords, etc. Then run query