Relational Algebra, SQL, Triggers, and Active Database

Relational Query Languages

- Languages for describing queries on a relational database
- *Structured Query Language* (SQL)
  - Predominant application-level query language
  - Declarative
- *Relational Algebra*
  - Intermediate language used within DBMS
  - Procedural

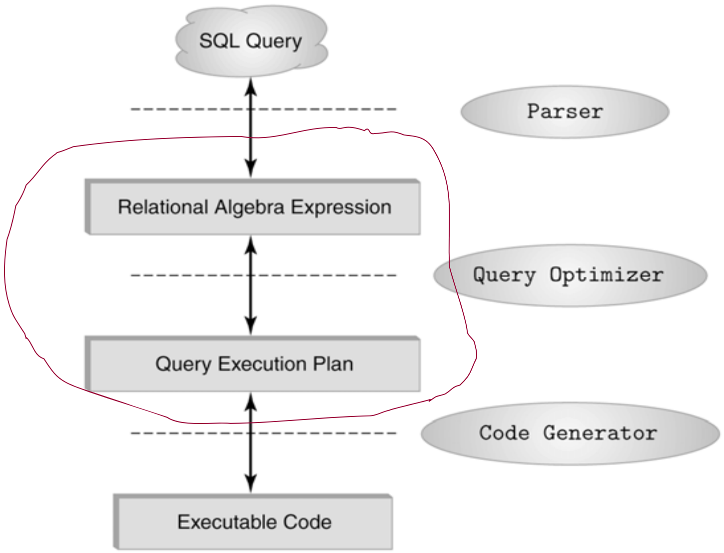
What is an Algebra?

- 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 are as 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*: set of relations
- *Basic operators*: select, project, union, set difference, Cartesian product
- *Derived operators*: set intersection, division, join
- *Procedural*: Relational expression specifies query by describing an algorithm (the sequence in which operators are applied) for determining the result of an expression

The Role of Relational Algebra in a DBMS



Schema for 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*)

Select Operator

- Produce table containing subset of rows of argument table satisfying condition

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

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

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

Selection Condition

- Operators: $<, \leq, \geq, >, =, \neq$
- Simple selection condition:
  - <*attribute*> *operator* <*constant*>
  - <*attribute*> *operator* <*attribute*>
- <*condition*> AND <*condition*>
- <*condition*> OR <*condition*>
- NOT <*condition*>

Selection Condition - Examples

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

Project Operator

- Produces table containing subset of columns of argument table

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

- 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

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

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

Project Operator

- 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

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

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

Result is a table (no duplicates); can have fewer tuples than the original

Expressions

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

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

- Relation is a set of tuples, so set operations should apply: $\cap$, $\cup$, $-$ (set difference)
- Result of combining two relations with a set operator is a relation => all its elements must be tuples having same structure
- Hence, scope of set operations 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*

Example

Tables:

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

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

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

But

π_{Name} (Person) and π_{Name} (Professor)

<u>are</u> union compatible so

π_{Name} (Person) - π_{Name} (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 and S need not be union compatible
- $R \times S$ is <u>expensive to compute</u>:
  - Factor of two in the size of each row
  - Quadratic in the number of rows

R:

A	*B*
x1	x2
x3	x4

S:

C	*D*
y1	y2
y3	y4

$R \times S$:

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

Renaming

- Result of expression evaluation is a relation
- 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 $A_1, A_2, \ldots A_n$ to the attributes of the *n* column relation produced by expression *expr* use

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

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

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

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

$$A_i\ oper\ B_i$$

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

The meaning is:

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

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*′.
  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

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

       In *R.A*, 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.

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

The join yields a table with attributes:
Employee.*Name*, Employee.*Id*, Employee.*Salary*, *MngrId*
Manager.*Name*, Manager.*Id*, Manager.*Salary*

Equijoin Join - Example

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

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

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

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

Transcript $\bowtie$ Teaching =

$\pi_{StudId,\ Transcript.CrsCode,\ Transcript.Sem,\ Grade,\ ProfId}$
(Transcript $\bowtie_{CrsCode=CrsCode \text{ AND } Sem=Sem}$ Teaching)
[*StudId, CrsCode, Sem, Grade, ProfId*]

Natural Join (cont'd)

- More generally:

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

where
attr-list = *attributes* (R) $\cup$ *attributes* (S)
(duplicates are eliminated) and *join-cond* has the form:

$A_1 = A_1$ AND … AND $A_n = A_n$

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:

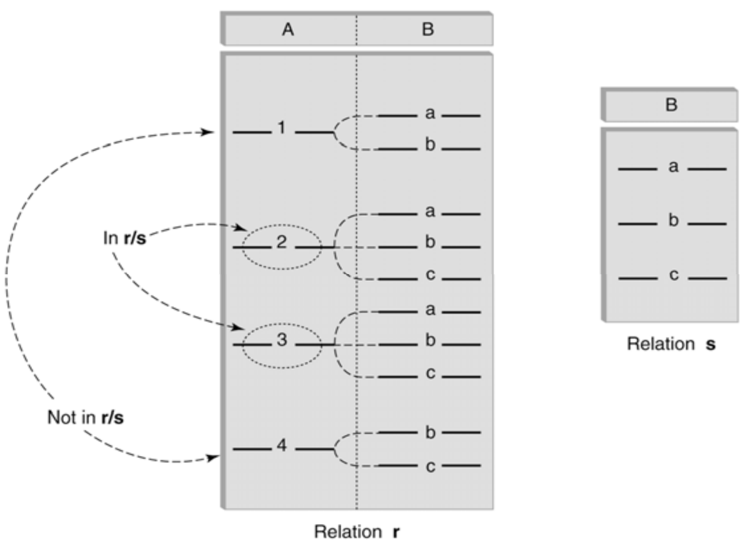
π_{StudId} ($\sigma_{CrsCode \neq CrsCode2}$ (
Transcript $\bowtie$
Transcript [*StudId, CrsCode2, Sem2, Grade2*]))

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

Division (cont'd)



Division - Example

- List the Ids of students who have passed *all* courses that were taught in spring 2000
- *Numerator*:
  - *StudId* and *CrsCode* for every course passed by every student:

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

- *Denominator*:
  - *CrsCode* of all courses taught in spring 2000

$$\pi_{CrsCode}\ (\sigma_{Semester='S2000'}\ (\text{Teaching})\ )$$

- Result is *numerator/denominator*

Schema for 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*)

Query Sublanguage of SQL

SELECT C.*CrsName*
FROM Course C
WHERE C.*DeptId* = 'CS'

- *Tuple variable* C ranges over rows of Course.
- Evaluation strategy:
  - FROM clause produces Cartesian product of listed tables
  - WHERE clause assigns rows to C in sequence and produces table containing only rows satisfying condition
  - SELECT clause retains listed columns
- Equivalent to: $\pi_{CrsName}\sigma_{DeptId=\text{'CS'}}(\text{Course})$

Join Queries

SELECT C.*CrsName*
FROM Course C, Teaching T
WHERE C.*CrsCode*=T.*CrsCode* AND T.*Semester*='S2000'

- List CS courses taught in S2000
- Tuple variables clarify meaning.
- Join condition "C.*CrsCode*=T.*CrsCode*"
  - relates facts to each other
- Selection condition " T.*Semester*='S2000' "
  - eliminates irrelevant rows
- Equivalent (using natural join) to:

$$\pi_{CrsName}(\text{Course} \bowtie \sigma_{Semester=\text{'S2000'}}(\text{Teaching}))$$
$$\pi_{CrsName}(\sigma_{Sem=\text{'S2000'}}(\text{Course} \bowtie \text{Teaching}))$$

Correspondence Between SQL and Relational Algebra

SELECT C.*CrsName*
FROM Course C, Teaching T
WHERE C.*CrsCode* = T.*CrsCode* AND T.*Semester* = 'S2000'

Also equivalent to:

$$\pi_{CrsName}\,\sigma_{C\_CrsCode=T\_CrsCode\ AND\ Semester=\text{'S2000'}}(\text{Course}\,[C\_CrsCode, DeptId, CrsName, Desc] \times \text{Teaching}\,[ProfId, T\_CrsCode, Semester])$$

- This is the simplest evaluation algorithm for SELECT.
- Relational algebra expressions are procedural.
  - Which of the two equivalent expressions is more easily evaluated?

Self-join Queries

Find 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 …..
```


Use of 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 CS Department and all professors that have taught CS courses

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

Nested Queries

List all courses that were not taught in S2000

```
SELECT C.CrsName
FROM Course C
WHERE C.CrsCode NOT IN
     (SELECT T.CrsCode      --subquery
      FROM Teaching T
      WHERE T.Sem = 'S2000')
```

Evaluation strategy: subquery evaluated once to produces set of courses taught in S2000. 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
           )
```

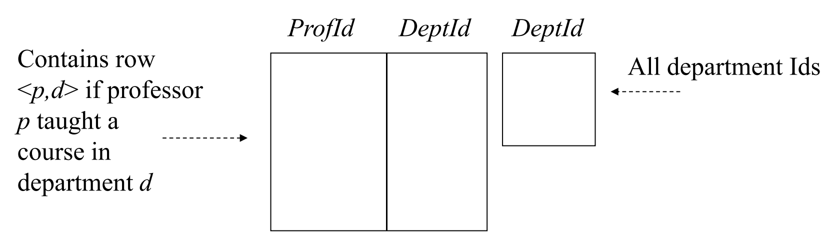

Correlated Nested Queries (con't)

- Tuple variables T and C are *local* to subquery
- Tuple variables 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 (at least) 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
  - Why does this involve division?



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

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 ⊇ B, or, equivalently, if B–A is empty

Division – SQL Solution

```
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)
- Not part of relational algebra (but not hard to add)

```
SELECT COUNT(*)          SELECT MAX (Salary)
FROM  Professor P        FROM  Employee E
```


Aggregates (cont'd)

Count the number of courses taught in S2000

```
SELECT COUNT (T.CrsCode)
FROM Teaching T
WHERE  T.Semester = 'S2000'
```

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

```
SELECT COUNT (DISTINCT T.CrsCode)
FROM Teaching T
WHERE  T.Semester = 'S2000'
```

Grouping

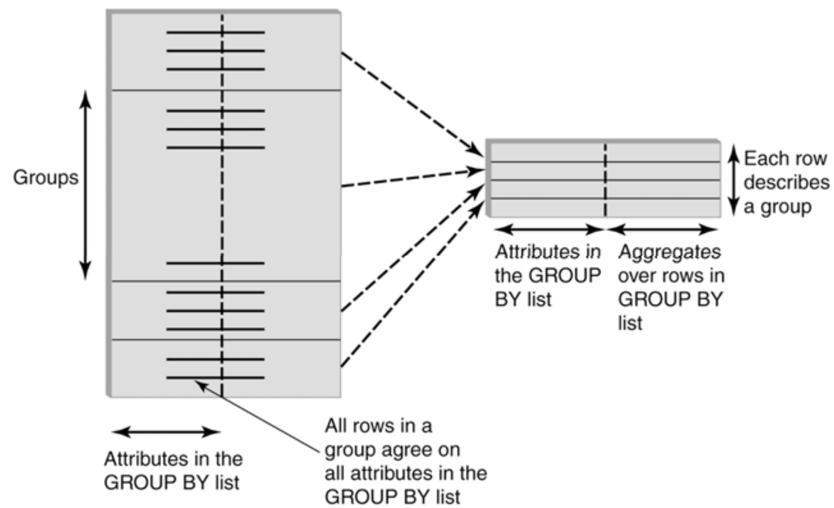
- But how do we compute the number of courses taught in S2000 *per professor*?
  - Strategy 1: Fire off a separate query for <u>each</u> professor:

```
SELECT  COUNT(T.CrsCode)
FROM    Teaching T
WHERE   T.Semester = 'S2000' AND T.ProfId = 123456789
```

    - Cumbersome
    - What if the number of professors changes? Add another query?
  - Strategy 2: define a special *grouping operator*:

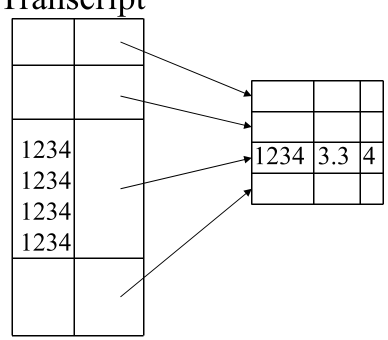
```
SELECT    T.ProfId, COUNT(T.CrsCode)
FROM      Teaching T
WHERE     T.Semester = 'S2000'
GROUP BY  T.ProfId
```


GROUP BY



GROUP BY - Example

Transcript



Attributes:
- student's *Id*
- avg grade
- number of courses

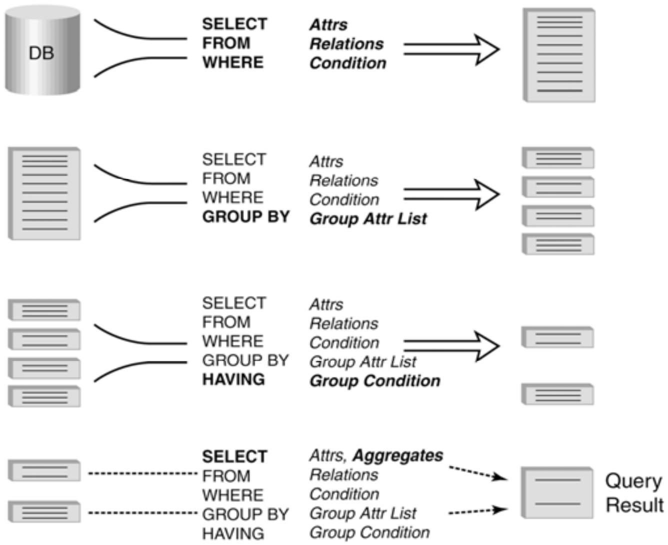
```
SELECT T.StudId, AVG(T.Grade), COUNT (*)
FROM Transcript T
GROUP BY T.StudId
```


HAVING Clause

- Eliminates 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

```
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
```

Evaluation of GroupBy with Having



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*

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

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

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*

WHERE T.*Grade* > COUNT (SELECT ….)
– *aggregate cannot be applied to result of* SELECT *statement*

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

As before:

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

Nulls

- *Conditions*: *x op y* (where *op* is <, >, <>, =, etc.) has value *unknown* (*U*) when either x or y is null
  - WHERE T.*cost* > T.*price*
- *Arithmetic expression*: x *op y* (where *op* is +, –, *, etc.) has value NULL if x or y is NULL
  - WHERE (T. *price*/T.*cost*) > 2
- *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'
```


Nulls (cont'd)

- 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

- Rows are discarded if WHERE condition is *F(alse)* or U(*nknown)*
- Ex: WHERE T.*CrsCode* = 'CS305' AND T.*Grade* > 2.5

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', 'S2000', 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 <> 'S2000'
```


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='S2000'
```

Updating Views - Problem 1

INSERT INTO CsReg (*StudId, CrsCode, Semester*)
VALUES (1111, 'CSE305', 'S2000')

- **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', 'S2000')

- **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 <u>*not*</u> <u>*uniquely*</u> 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 (cont'd)

- 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*

Triggers and Active Databases

Trigger Overview

- Element of the database schema
- General form:
  ON <*event*> IF <*condition*> THEN <*action*>
  - *Event*- request to execute database operation
  - *Condition* - predicate evaluated on database state
  - *Action* – execution of procedure that might involve database updates
- Example:
  ON updating maximum course enrollment
  IF number registered > new max enrollment limit
  THEN deregister students using LIFO policy

Trigger Details

- **Activation** - Occurrence of the *event*
- **Consideration** - The point, after activation, when *condition* is evaluated
  - Immediate or deferred (when the transaction requests to commit)
  - *Condition* might refer to both the state before and the state after *event* occurs

Trigger Details

- **Execution** – point at which *action* occurs
  - With deferred consideration, execution is also deferred
  - With immediate consideration, execution can occur immediately after consideration or it can be deferred
    - If execution is immediate, execution can occur before, after, or instead of triggering event.
    - Before triggers adapt naturally to maintaining integrity constraints: violation results in rejection of event.

Trigger Details

- **Granularity**
  - *Row-level granularity*: change of a single row is an event (a single UPDATE statement might result in multiple events)
  - *Statement-level granularity*: events are statements (a single UPDATE statement that changes multiple rows is a single event).

Trigger Details

- **Multiple Triggers**
  - How should multiple triggers activated by a single event be handled?
    - Evaluate one condition at a time and if true immediately execute action or
    - Evaluate all conditions, then execute actions
  - The execution of an action can affect the truth of a subsequently evaluated condition so the choice is significant.

Triggers in SQL:1999

- **Events**: INSERT, DELETE, or UPDATE statements or changes to individual rows caused by these statements
- **Condition**: Anything that is allowed in a WHERE clause
- **Action**: An individual SQL statement or a program written in the language of Procedural Stored Modules (PSM) (which can contain embedded SQL statements)

Triggers in SQL:1999

- **Consideration**: *Immediate*
  - Condition can refer to both the state of the affected row or table before *and* after the event occurs
- **Execution**: *Immediate* – can be before or after the execution of the triggering event
  - Action of before trigger cannot modify the database
- **Granularity**: Both *row-level* and *statement-level*

Before Trigger Example

(row granularity)

Check that enrollment ≤ limit

```
CREATE TRIGGER Max_EnrollCheck
  BEFORE INSERT ON Transcript
      REFERENCING NEW AS N   --row to be added
  FOR EACH ROW
  WHEN
  ((SELECT  COUNT (T.StudId) FROM Transcript T
     WHERE  T.CrsCode = N.CrsCode
                 AND T.Semester = N.Semester)
   >=
   (SELECT C.MaxEnroll FROM Course C
     WHERE C.CrsCode = N.CrsCode ))
  ABORT TRANSACTION
```


After Trigger Example

(row granularity)

No salary raises greater than 5%

```
CREATE TRIGGER LimitSalaryRaise
  AFTER UPDATE OF Salary ON Employee
  REFERENCING OLD AS O
                    NEW AS N
  FOR EACH ROW
  WHEN (N.Salary - O.Salary > 0.05 * O.Salary)
     UPDATE Employee         -- action
     SET Salary = 1.05 *  O.Salary
     WHERE Id = O.Id
```

Note: The action itself is a triggering event (but in this case a chain reaction is not possible)

After Trigger Example

(statement granularity)

Keep track of salary averages in the log

```
CREATE TRIGGER RecordNewAverage
  AFTER UPDATE OF Salary ON Employee
  FOR EACH STATEMENT
    INSERT INTO Log
    VALUES  (CURRENT_DATE,
                  SELECT AVG (Salary)
                  FROM  Employee)
```