Lecture 06 (Chapter 5)
Relational Algebra: Under the Hood of SQL

Lecturer: Sael Lee

Slide adapted from the author’s and Dr. Ilchul Yoon’s slides.
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

- SQL Query
- Relational Algebra Expression
- Query Execution Plan
- Executable Code
- Parser
- Query Optimizer
- Code Generator
Select Operator

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

\[ \sigma_{condition} (relation) \]

- Example:

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>stamps</td>
</tr>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>coins</td>
</tr>
<tr>
<td>5556</td>
<td>Mary</td>
<td>7 Lake Dr</td>
<td>hiking</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
<td>5 Pine St</td>
<td>stamps</td>
</tr>
</tbody>
</table>

\[ \sigma_{Hobby='stamps'} (Person) \]

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>stamps</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
<td>5 Pine St</td>
<td>stamps</td>
</tr>
</tbody>
</table>
Selection Condition

- Operators: <, ≤, ≥, >, =, ≠
- Simple selection condition:
  - `<attribute>` operator `<constant>`
  - `<attribute>` operator `<attribute>`
- `<condition>` AND `<condition>`
- `<condition>` OR `<condition>`
- NOT `<condition>`
Selection Condition - Examples

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

- Produces table containing subset of columns of argument table

\[ \pi_{\text{attribute list}}(\text{relation}) \]

- Example:

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>stamps</td>
</tr>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>coins</td>
</tr>
<tr>
<td>5556</td>
<td>Mary</td>
<td>7 Lake Dr</td>
<td>hiking</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
<td>5 Pine St</td>
<td>stamps</td>
</tr>
</tbody>
</table>

\[ \pi_{\text{Name, Hobby}}(\text{Person}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>stamps</td>
</tr>
<tr>
<td>John</td>
<td>coins</td>
</tr>
<tr>
<td>Mary</td>
<td>hiking</td>
</tr>
<tr>
<td>Bart</td>
<td>stamps</td>
</tr>
</tbody>
</table>
Project Operator

Example:

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>stamps</td>
</tr>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>coins</td>
</tr>
<tr>
<td>5556</td>
<td>Mary</td>
<td>7 Lake Dr</td>
<td>hiking</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
<td>5 Pine St</td>
<td>stamps</td>
</tr>
</tbody>
</table>

\[ \pi_{\text{Name,Address}}(\text{Person}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>123 Main</td>
</tr>
<tr>
<td>Mary</td>
<td>7 Lake Dr</td>
</tr>
<tr>
<td>Bart</td>
<td>5 Pine St</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>stamps</td>
</tr>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>coins</td>
</tr>
<tr>
<td>5556</td>
<td>Mary</td>
<td>7 Lake Dr</td>
<td>hiking</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
<td>5 Pine St</td>
<td>stamps</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
</tr>
</tbody>
</table>

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
Two relations are union compatible if
- Both have the 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 not union compatible.

But

\[ \pi_{Name} (\text{Person}) \text{ and } \pi_{Name} (\text{Professor}) \]

are 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 $S$
  - $R$ and $S$ need not be union compatible
- **$R \times S$ is expensive to compute:**
  - Factor of two in the size of each row
  - Quadratic in the number of rows
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 \((\text{StudId, CrsCode, Semester, Grade})\)
Teaching \((\text{ProfId, CrsCode, Semester})\)

\[\pi_{\text{StudId, CrsCode}} (\text{Transcript})[\text{StudId, CrsCode1}] \times \pi_{\text{ProfId, CrsCode}} (\text{Teaching}) [\text{ProfId, CrsCode2}]\]

This is a relation with 4 attributes:
\[\text{StudId, CrsCode1, ProfId, CrsCode2}\]
Derived Operation: Join (theta-join)

- A (general or theta) join of $R$ and $S$ is the expression
  \[ R \bowtie_{\text{join-condition}} S \]
  where join-condition is a conjunction of terms:
  \[ A_i \text{ oper } B_i \]
  in which $A_i$ is an attribute of $R$; $B_i$ is an attribute of $S$; and
  oper is one of $=, <, >, \geq, \neq, \leq$.

- The meaning is:
  \[ \sigma_{\text{join-condition'}} (R \times S) \]
  where join-condition and join-condition' are the same, except for possible renaming of attributes.
Join and Renaming

**Problem:**
- R and S might have attributes with the same name – in which case the Cartesian product is not defined.

**Solutions:**
- Rename attributes prior to forming the product and use new names in join-condition.
- Qualify common attribute names with relation names (thereby disambiguating the names).
  - e.g., Transcript.CrsCode or Teaching.CrsCode.
- This solution is nice, but doesn’t always work: consider
  - R join_condition R
  - In R.A, how do we know which R is meant?
Theta Join – Example

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

\[ \pi_{\text{Employee.Name}} (\text{Employee} \bowtie \text{Manager} \\
\quad \text{E.MngrId} = \text{M.Id} \quad \text{AND} \quad \text{E.Salary} > \text{M.Salary} ) \]

**The join yields a table with attributes:**
- **Employee.Name, Employee.Id, Employee.Salary, MngrlId, Manager.Name, Manager.Id, Manager.Salary**
Equijoin Join - Example

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

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

**Student**

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Addr</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>John</td>
<td>.....</td>
<td>.....</td>
</tr>
<tr>
<td>222</td>
<td>Mary</td>
<td>.....</td>
<td>.....</td>
</tr>
<tr>
<td>333</td>
<td>Bill</td>
<td>.....</td>
<td>.....</td>
</tr>
<tr>
<td>444</td>
<td>Joe</td>
<td>.....</td>
<td>.....</td>
</tr>
</tbody>
</table>

**Transcript**

<table>
<thead>
<tr>
<th>StudId</th>
<th>CrsCode</th>
<th>Sem</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>CSE305</td>
<td>S00</td>
<td>B</td>
</tr>
<tr>
<td>222</td>
<td>CSE306</td>
<td>S99</td>
<td>A</td>
</tr>
<tr>
<td>333</td>
<td>CSE304</td>
<td>F99</td>
<td>A</td>
</tr>
</tbody>
</table>

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

Mary  CSE306
Bill   CSE304
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

\[
\text{Transcript} (\text{StudId}, \text{CrsCode}, \text{Sem}, \text{Grade}) \\
\text{Teaching} (\text{ProfId}, \text{CrsCode}, \text{Sem})
\]

\[
\text{Transcript} \bowtie \text{Teaching} = \\
\pi_{\text{StudId}, \text{Transcript.CrsCode}, \text{Transcript.Sem}, \text{Grade}, \text{ProfId}} \\
( \text{Transcript} \bowtie_{\text{CrsCode} = \text{CrsCode} \text{ AND Sem} = \text{Sem}} \text{Teaching} ) \\
\left[ \text{StudId}, \text{CrsCode}, \text{Sem}, \text{Grade}, \text{ProfId} \right]
\]
Natural Join (cont’d)

- More generally:

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

where

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

(duplicates are eliminated) and

(2) \text{join-cond} has the form:

\[ A_1 = A_1 \text{ AND } \ldots \text{ AND } A_n = A_n \]

where

\( \{A_1 \ldots A_n\} = \text{attributes}(R) \cap \text{attributes}(S) \)
Natural Join Example

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

\[ \pi_{\text{StudId}} \left( \sigma_{\text{CrsCode} \neq \text{CrsCode2}} \left( \text{Transcript} \otimes \text{Transcript} \left[ \text{StudId}, \text{CrsCode2}, \text{Sem2}, \text{Grade2} \right] \right) \right) \]

We don’t want to join on \text{CrsCode}, \text{Sem}, and \text{Grade} attributes, hence renaming!
Outer Join

- Three types
  - Left outer join / Right outer join / Full outer join
- Given two relations $r$ and $s$, the tuples in $r \bowtie_{\text{outer}} s$ consist of three categories
  1. The tuples that appear in the regular join of $r$ and $s$, $r \bowtie_{\text{cond}} s$
  2. The tuples of $r$ that do not join with any tuple in $s$
  3. The tuples of $s$ that do not join with any tuple in $r$
- For left outer join, $1 \cup 2$
- For right outer join, $1 \cup 3$
- For full outer join, $1 \cup 2 \cup 3$
Lecture 07 (Chapter 5)
Relational Algebra: Under the Hood of SQL

Lecturer: Sael Lee

Slide adapted from the author’s and Dr. Ilchul Yoon’s slides.
Division

- Goal: Produce the tuples in one relation, $r$, that match all tuples in another relation, $s$
  - $r (A_1, ... A_n, B_1, ... B_m)$
  - $s (B_1, ..., B_m)$
  - $r/s$, with attributes $A_1, ..., A_n$, 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)

Relation $r$

Relation $s$

$A$  $B$

1. $a$
   $b$

2. $a$
   $b$
   $c$

3. $a$
   $b$
   $c$

4. $b$
   $c$

In $r/s$

Not in $r/s$
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

\[
T_1 = \pi_A (R) \times S \\
T_2 = \pi_A (T_1 \setminus R) \\
T_3 = \pi_A (R) \setminus T_2
\]
Division - Example

- List the Ids of students who have passed all courses that were taught in spring 2006

- **Numerator:**
  - *StudId* and *CrsCode* for every course passed by every student:
    \[ \pi_{\text{StudId}, \text{CrsCode}} (\sigma_{\text{Grade} \neq \text{F}} (\text{Transcript})) \]

- **Denominator:**
  - *CrsCode* of all courses taught in spring 2006
    \[ \pi_{\text{CrsCode}} (\sigma_{\text{Semester}=\text{S2006}} (\text{Teaching})) \]

- Result is numerator/denominator
Example

Suppose we want to know, from the Movies relation, “What are the titles and years of movies made by Fox that are at least 100 minutes long?”

Movies(title, year, length, genre, studioName, producerC#)
MoviewStar(name, address gender, DOB)
StarIn(movieTitle, movieYear, starName)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, PresC#)
Administrivia

- No class on 3/31 (Tuesday) and 4/2 (Thursday)
  - Makeup class: TBA

- Term Project requirements
  - General requirements and Journal/conference attributes are posted in Bb.
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.
- **An** 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: $\pi_{\text{CrsName}} \sigma_{\text{DeptId} = 'CS'}(\text{Course})$
Join Queries

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

- List 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:
  
  $\pi_{\text{CrsName}}(\sigma_{\text{Sem}='S2000'}(\text{Teaching} \bowtie \text{Teaching}))$

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

\[ \pi_{\text{CrsName}} \sigma_{\text{C.CrsCode}=\text{T.CrsCode AND Semester='S2000'}}(\text{Course} \left[ \text{C.CrsCode, Deptld, CrsName, Desc} \right] \times \text{Teaching} \left[ \text{Profld, T.CrsCode, Semester} \right]) \]

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:

  ```sql
  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:

  ```sql
  π_{ProfId} (σ_{T1.CrsCode<>T2.CrsCode} (Teaching[ProfId, T1.CrsCode, Semester]
  ⨝ Teaching[ProfId, T2.CrsCode, Semester]))
  ```
Duplicates

- Duplicate rows are 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

\[
\text{(SELECT } P.\text{Name} \\
\text{FROM Professor P, Teaching T} \\
\text{WHERE P.Id=T.ProfId AND T.CrsCode LIKE 'CS%')} \text{ UNION} \\
\text{(SELECT } P.\text{Name} \\
\text{FROM Professor P} \\
\text{WHERE P.DeptId = 'CS'})
\]
Nested Queries

- List all courses that were not taught in S2000

```sql
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 \(<\text{prof}, \text{dept}>\) if \text{prof} has taught a course in \text{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*
EXISTS Operator

- Simply, used to check if a nested subquery returns no answers
- Example
  - Find all students who never took a computer science course

\[
\begin{align*}
\text{SELECT} & \quad S.\text{Id} \\
\text{FROM} & \quad \text{Student S} \\
\text{WHERE} & \quad \text{NOT EXISTS (} \\
& \quad \quad \quad \quad \quad -- \text{All CS courses taken by S.Id} \\
& \quad \quad \quad \quad \quad \text{SELECT} \quad T.\text{CrsCode} \\
& \quad \quad \quad \quad \quad \text{FROM} \quad \text{Transcript T} \\
& \quad \quad \quad \quad \quad \text{WHERE} \quad T.\text{CrsCode} \text{ LIKE ‘CS%’ AND} \\
& \quad \quad \quad \quad \quad T.\text{StudID} = S.\text{Id} \quad )
\end{align*}
\]
Division in SQL

- **Query type**: Find the subset of items in one set that are related to *all* items in another set
- **Example**
  - Find professor IDs who taught courses in all departments
  - Why does this involve division?

\[
\pi_{\text{ProfId},\text{DeptId}}(\text{Teaching} \Join \text{Course}) / \pi_{\text{DeptId}}(\text{Department})
\]
Division in SQL

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

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

- Is there a student in the university whose GPA is higher than that of all junior students?

```sql
SELECT S.Id
FROM Student S
WHERE S.GPA > ALL ( SELECT S.GPA
FROM Student S
WHERE S.Status = 'junior' )
```

- What happens if we replace > ALL with >= ANY?
Nested Query in the FROM clause

- Use nested query in the FROM clause and rename it
- For example
  - *Find the students who took a course from every professor in the CS department*

```sql
SELECT S.Id
FROM Student S
EXCEPT
SELECT S.Id
FROM Student S,
     ( SELECT P.Id
         FROM Professor P
         WHERE P.Dept = 'CS' ) AS C
WHERE C.ProfId NOT IN
     ( SELECT T.ProfId
         FROM Teaching T, Transcript R
         WHERE T.CrsCode = R.CrsCode AND
               T.Semester = R.Semester AND
               S.Id = R.StudId )
```
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)

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

- Do not mix aggregate and an attribute in the SELECT clause

```sql
SELECT COUNT(*), S.Id FROM Student S
WHERE S.Name = 'JohnDoe'
```

- Aggregate functions can’t be used in the WHERE clause
Aggregates (cont’d)

- Count the number of courses taught in S2000

\[
\text{SELECT COUNT (T.CrsCode) FROM Teaching T WHERE T.Semester = 'S2000'}
\]

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

\[
\text{SELECT COUNT (DISTINCT T.CrsCode) FROM Teaching T WHERE T.Semester = 'S2000'}
\]
Grouping

- But how do we compute the number of courses taught in S2000 per professor?

  - Strategy 1: Fire off a separate query for each professor:
    ```sql
    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:
    ```sql
    SELECT T.ProfId, COUNT(T.CrsCode)
    FROM Teaching T
    WHERE T.Semester = 'S2000'
    GROUP BY T.ProfId
    ```
GROUP BY

Each row describes a group

Attributes in the GROUP BY list

Aggregates over rows in GROUP BY list

All rows in a group agree on all attributes in the GROUP BY list

Groups
GROUP BY - Example

<table>
<thead>
<tr>
<th>Transcript</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
</tr>
<tr>
<td>1234</td>
</tr>
</tbody>
</table>

**Attributes:**
- student’s Id
- AVG grade
- number of courses

```sql
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

```sql
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
```
Query Evaluation with Aggregate Functions

```
SELECT Attrs FROM WHERE Relations Condition

SELECT Attrs FROM WHERE Relations Condition GROUP BY Group Attr List

SELECT Attrs FROM WHERE Relations Condition GROUP BY Group Attr List HAVING Group Condition

SELECT Attrs, Aggregates FROM WHERE Relations Condition GROUP BY Group Attr List HAVING Group Condition
```

Query Result
Example

- Output the id and name of all seniors on the Dean’s List (average grade over 3.5, and more than 90 credits.)

```sql
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.
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

```sql
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 StudIdc
```

- 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
JOIN Expressions in the FROM Clause

- Called ‘table expressions’
- Format
  - Table1 [NATURAL] [INNER|FULL|LEFT|OUTER] JOIN table 2 [ON condition]
- List average grade for every student in the database

SELECT S.Name, AVG(S.Grade)
FROM Student LEFT JOIN Transcript
ON Student.Id = Transcript.StudId AS S
GROUP BY S.Id

VS.

SELECT S.Name, AVG(T.Grade)
FROM Student S, Transcript T
WHERE S.Id = T.StudId
GROUP BY S.Id
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
CREATE VIEW CumGpa (StudId, Cum) AS
SELECT T.StudId, AVG (T.Grade)
FROM Transcript T
GROUP BY T.StudId

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 are 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

- CREATE VIEW PartOfTranscript (StudId, CrsCode, Semester) AS
  SELECT T.StudId, T.CrsCode, T.Semester -- limit columns
  FROM Transcript T
  WHERE T.Semester = 'S2000' -- 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, if SQL allowed attribute lists in GRANT SELECT
View Benefits (cont’d)

- **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
  - Ease of use and learning
  - Security
  - Logical data independence

- A view is *similar in many ways to a subroutine* in standard programming
  - Can be reused in multiple queries
Materialized Views

- Cached view – caching if popular with many queries
  - Dramatic reduction of query response time
  - Expensive update operation
  - Adding/removing tuples in base relations may (or may not) affect view

![Table Example]

**FIGURE 3.9** Contents of the view defined by SQL statement (3.5).
Materialized Views (Con’d)

- View cache maintenance are expensive.
  - Incremental re-compute considering changes

- View cache is very important in data warehousing
  - *Data warehouse is an (infrequently updated) database that typically consist of complex materialized views of the data stored in a separate production database.*
  - Optimized for querying, not transaction processing

CREATE MATERIALIZED VIEW PROFSTUD(Prof, Stud)  
   BUILD IMMEDIATE  
   REFRESH FAST ON COMMIT  
   ENABLE QUERY REWRITE  
AS  
   SELECT T.ProfId, R.StudId)  
   FROM Transcript R, Teaching T  
   WHERE .....  

Oracle Example
CSE 305 / CSE532

Lecture 08 (Chapter 5)
Relational Algebra: Under the Hood of SQL

Lecturer: Sael Lee

Slide adapted from the author’s and Dr. Ilchul Yoon’s slides.
Nulls and 3-valued logic

- **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**

```sql
SELECT COUNT (T.CrsCode), AVG (T.Grade) FROM Transcript T WHERE T.StudId = '1234'
```
Nulls (cont’d)

- WHERE clause uses a three-valued logic – $T$, $F$, $U(\text{undefined})$ – to filter rows. Portion of truth table:

\[
\begin{array}{cccc}
\text{C1} & \text{C2} & \text{C1 AND C2} & \text{C1 OR C2} \\
T & U & U & T \\
F & U & F & U \\
U & U & U & U \\
\end{array}
\]

- Rows are discarded if WHERE condition is $F(\text{alse})$ or $U(\text{unknown})$
  - e.g., WHERE $T.\text{CrsCode} = \text{‘CS305’}$ AND $T.\text{Grade} > 2.5$
- If a CHECK clause evaluates to unknown, the integrity constraints are considered to be observed.
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

```sql
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 and Update

- **DELETE** is 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

  ```sql
  DELETE FROM Transcript T
  WHERE T.Grade IS NULL AND T.Semester <> 'S2000'
  ```

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

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

Question: What value should be placed in attributes of underlying table that have been projected out (e.g., Grade)?

Answer: Simple. NULL (assuming null allowed in the missing attribute) or DEFAULT

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

- 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 - Problem 3 (Cont’d)

- What to do at deleting <101202303, 123454321>>
  - Delete two rows from Teaching?
  - Delete two rows from Transcript?
  - Delete one with CS315 from Teaching and one with CS305 Transcript from Transcript? Or, CS305 from Teaching and CS315 from Transcript?

![Table Image]
Updatable Views - Restrictions

- Updatable views are restricted to those in which
  - No Cartesian product in FROM clause
  - No aggregates, GROUP BY, HAVING, set operation
  - No expressions, DISTINCT keyword in the SELECT clause
  - ...

For example, if we allowed:

```sql
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:

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