Admiral Yoon
Assistant Professor
State University of New York, Korea

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: \( \pi_{\text{CrsName}} \sigma_{\text{DeptId}=\text{‘CS’}}(\text{Course}) \)
Join Queries

- 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:

\[
\pi_{\text{CrsName}}( \sigma_{\text{Sem}=\text{S2000}} (\text{Teaching}) ) \\
\pi_{\text{CrsName}} (\sigma_{\text{Sem}=\text{S2000}} (\text{Course} \bowtie \text{Teaching}))
\]

Correspondence Between SQL and Relational Algebra

- Also equivalent to:

\[
\pi_{\text{CrsName}} (\sigma_{C.CrsCode = T.CrsCode \land T.Semester = 'S2000'} (\text{Course} [C.CrsCode, \text{DeptId}, \text{CrsName}, \text{Desc}] \\
\bowtie \text{Teaching} [\text{ProfId}, T.CrsCode, \text{Semester}]))
\]

- 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:
  
  ```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
  \pi_{Profid}(\sigma_{T1.CrsCode=T2.CrsCode}(Teaching[Profid, T1.CrsCode, Semester] \bowtie 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:

  ```sql
  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**

**EXISTS Operator**

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

```
SELECT S.Id
FROM Student S
WHERE NOT EXISTS (  
  -- All CS courses taken by S.Id
  SELECT T.CrsCode
  FROM Transcript T
  WHERE T.CrsCode LIKE 'CS%' AND  
  T.StudID = S.Id )
```
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?

Contains row \(<p,d>\) if professor \(p\) taught a course in department \(d\)

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

**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 \(A \supseteq B\), or, equivalently, if \(B - A\) is empty
### Division – SQL Solution

```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)
```

### 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

```
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 )
```

For example
- Find the students who took a course from every professor in the CS department

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

- Do not mix aggregate and an attribute in the SELECT

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

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

Aggregate over rows in GROUP BY list

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

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

```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
Example

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

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

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

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

  vs.

  ```sql
  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
**View – Usage Example**

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**
  
  ```sql
  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:
  ```sql
  GRANT SELECT ON PartOfTranscript TO joe
  ```

- This would have been analogous to:
  ```sql
  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

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*