CS 5614: (Big) Data Management Systems

B. Aditya Prakash
Lecture #3: SQL and Relational Algebra---More operators, DDL and Views
Aggregate functions

- find avg grade, across all students
  
  select ??
  
  from takes

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-413</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>15-413</td>
<td>3</td>
</tr>
</tbody>
</table>
Aggregate functions

- find avg grade, across all students
  
  select avg(grade)
  
  from takes

- result: a single number

- Which other functions?

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-413</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>15-413</td>
<td>3</td>
</tr>
</tbody>
</table>
Aggregate Operators

- **COUNT (*)**
- **COUNT ([DISTINCT] A)**
  - A is a column
- **SUM ([DISTINCT] A)**
- **AVG ([DISTINCT] A)**
- **MAX (A)**
- **MIN (A)**
Aggregate functions

- find total number of enrollments
  
  select count(*)

  from takes

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-413</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>15-413</td>
<td>3</td>
</tr>
</tbody>
</table>
Aggregate functions

- find total number of students in 15-413
  
  select count(*)
  
  from takes
  
  where c-id="15-413"

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-413</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>15-413</td>
<td>3</td>
</tr>
</tbody>
</table>
Find name and age of the oldest sailor(s)

```
SELECT S.sname, MAX (S.age)
FROM Sailors S
```

- This is illegal, but why?
  - Cannot combine a column with a value

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age = (SELECT MAX (S2.age) FROM Sailors S2)
```
GROUP BY and HAVING

- So far, aggregate operators are applied to all (qualifying) tuples.
  - Can we apply them to each of several groups of tuples?
- Example: find the age of the youngest sailor for each rating level.
  - In general, we don’t know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this:

  For $i = 1, 2, \ldots, 10$:  
  \[
  \text{SELECT \ MIN (S.age) FROM \ Sailors \ S WHERE \ S.rating = i}
  \]
Find the age of the youngest sailor for each rating level

```
SELECT S.rating, MIN (S.age) as age
FROM Sailors S
GROUP BY S.rating
```

(1) The sailors tuples are put into “same rating” groups.

(2) Compute the Minimum age for each rating group.

<table>
<thead>
<tr>
<th>Sid</th>
<th>Sname</th>
<th>Rating</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>85</td>
<td>Art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>32</td>
<td>Andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rating</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Find the age of the youngest sailor for each rating level that has at least 2 members

```sql
SELECT S.rating, MIN(S.age) as minage
FROM Sailors S
GROUP BY S.rating
HAVING COUNT(*) > 1
```

1. The sailors tuples are put into “same rating” groups.
2. Eliminate groups that have < 2 members.
3. Compute the Minimum age for each rating group.

<table>
<thead>
<tr>
<th>Sid</th>
<th>Sname</th>
<th>Rating</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>85</td>
<td>Art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>32</td>
<td>Andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rating</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Drill

- find total number of students in each course

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-413</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>15-413</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c-id</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-413</td>
<td>2</td>
</tr>
</tbody>
</table>
Drill

- find total number of students in each course

select c-id, count(*)
from takes

group by c-id

order by c-id

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-413</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>15-413</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c-id</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-413</td>
<td>2</td>
</tr>
</tbody>
</table>
Drill

- find total number of students in each course, and sort by count, decreasing

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-413</td>
<td>4</td>
</tr>
<tr>
<td>234</td>
<td>15-413</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c-id</th>
<th>pop</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-413</td>
<td>2</td>
</tr>
</tbody>
</table>
Drill

- find total number of students in each course, and sort by count, decreasing

select c-id, count(*) as pop
from takes
group by c-id
order by pop desc
Queries With \textit{GROUP BY} and \textit{HAVING}

\begin{verbatim}
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
\end{verbatim}

- The \textit{target-list} contains (i) attribute names (ii) terms with aggregate operations (e.g., \textit{AVG} (\textit{S.age})).
- The attribute list (e.g., \textit{S.rating}) in \textit{target-list} must be in \textit{grouping-list}.
- The attributes in group-qualification must be in \textit{grouping-list}.

\begin{verbatim}
SELECT S.rating, \text{MIN} (S.age) as age
FROM Sailors S
GROUP BY S.rating
HAVING S.rating > 5
\end{verbatim}
Motivation for Subqueries

- Find the name of the professor who teaches “CS 4604.”

```sql
SELECT Name
FROM Professors, Teach
WHERE (PID = ProfessorPID) AND (Number = '4604') AND (DeptName = 'CS');
```

- Do we need to take the natural join of two big relations just to get a relation with one tuple?
- Can we rewrite the query without using a join?
Nesting

- A query can be put inside another query
- Most commonly in the WHERE clause
- Sometimes in the FROM clause (depending on the software)
- This subquery is executed first (if possible)
Subquery Example

- Find the name of the professor who teaches “CS 4604.”

SELECT Name
FROM Professors
WHERE PID = (SELECT ProfessorPID
FROM Teach
WHERE Number = 4604
AND DeptName = 'CS');
Subquery Example

- Find the name of the professor who teaches “CS 4604.”

```
SELECT Name
FROM Professors
WHERE PID =
    (SELECT ProfessorPID
     FROM Teach
     WHERE (Number = 4604) AND (DeptName = 'CS'))
```

- When using =, the subquery must return a single tuple
nested subqueries

- find student record with highest ssn
  
  select *
  from student
  where ssn is
  
  greater than every other ssn
nested subqueries

- find student record with highest ssn

\[
\text{select * from student where ssn \ greater \ than \ every select ssn from student}
\]
nested subqueries

- find student record with highest ssn

```sql
select *
from student
where ssn > all (select ssn from student)
```

almost correct
nested subqueries

- find student record with highest ssn
  select *
  from student
  where ssn >= all (select ssn from student)
Conditions Involving Relations

- SQL includes a number of operators that apply to a relation and produce a boolean result.
- These operators are very useful to apply on results of sub-queries.
Conditions Involving Relations

- Let R be a relation and t be a tuple with the same set of attributes.
  - **EXISTS** R is true if and only if R contains at least one tuple.
  - t **IN** R is true if and only if t equals a tuple in R.
  - t > **ALL** R is true if and only if R is unary (has one attribute) and t is greater than every value in R.
    - Can use any of the other five comparison operators.
    - If we use <>, R need not be unary.
  - t > **ANY** R (which is unary) is true if and only if t is greater than at least one value in R.
- We can use **NOT** to negate EXISTS, ALL, and ANY.
Subqueries Using Conditions

- Find the departments of the courses taken by the student with name ‘Suri’.

```
SELECT DeptName
FROM Take
WHERE StudentPID IN
  ( SELECT PID
    FROM Students
    WHERE (Name = 'Suri')
  );
```
Correlated vs Uncorrelated

- The previous subqueries did not depend on anything outside the subquery
  - ...and thus need to be executed just once.
  - These are called **uncorrelated**.

- A **correlated** subquery depends on data from the outer query
  - ... and thus has to be executed for each row of the outer table(s)
Correlated Subqueries

- Find course names that have been used for two or more courses.

```sql
SELECT CourseName
FROM Courses AS First
WHERE CourseName IN
  (SELECT CourseName
   FROM Courses
   WHERE (Number <> First.Number)
   AND (DeptName <> First.DeptName))
```

Prakash 2014
Evaluating Correlated Subqueries

SELECT CourseName
FROM Courses AS First
WHERE CourseName IN
    (SELECT CourseName
     FROM Courses
     WHERE (Number <> First.Number)
     AND (DeptName <> First.DeptName))

- Evaluate query by looping over tuples of First, and for each tuple evaluate the subquery.
- Scoping rules: an attribute in a subquery belongs to one of the tuple variables in that subquery’s FROM clause, or to the immediately surrounding subquery, and so on.
Subqueries in FROM clauses

- Can use a subquery as a relation in a FROM clause.
- We must give such a relation an alias using the AS keyword.
- Let us find different ways of writing the query “Find the names of Professors who have taught the student whose first name is ‘Suri’."
- The old way:

SELECT Professors.Name
FROM Professors, Take, Teach, Students
WHERE (Professors.PID = Teach.ProfessorPID)
  AND (Teach.CourseNumber = Take.CourseNumber)
  AND (Teach.DeptName = Take.DeptName)
  AND (Take.StudentPID = Student.PID)
  AND (Student.Name = 'Suri %');
“Find the names of (Professors who have taught (courses taken by (student with first name ‘Suri’))).”

SELECT Name
FROM Professors
WHERE PID IN
  (SELECT ProfessorPID
   FROM Teach
   WHERE (Number, DeptName) IN
     (SELECT Number, DeptName
      FROM Take, Students
      WHERE (StudentPID = PID) AND
      (Students.Name = ‘Suri%’)));
Unrolling it further

- SELECT Name
  FROM Professors
  WHERE PID IN
    (SELECT ProfessorPID
     FROM Teach
     WHERE (Number, DeptName) IN
       (SELECT Number, DeptName
        FROM Take
        WHERE StudentPID IN
          (SELECT PID
           FROM Students
           WHERE Name = 'Suri %')));
Extended Joins

so far: ‘INNER’ joins, eg:

\begin{verbatim}
select ssn, c-name 
from takes, class 
where takes.c-id = class.c-id
\end{verbatim}
Joins

Equivalently:

```
select ssn, c-name
from takes join class on takes.c-id = class.c-id
```
Outerjoin

- Suppose we have: $R \bowtie_C S$

- A tuple of $R$ that has no tuple of $S$ with which it joins is said to be *dangling*.
  - Similarly for a tuple of $S$.

- Outerjoin preserves dangling tuples by padding them with a special NULL symbol in the result.
Example: Outerjoin

\[ R = \begin{pmatrix} 1 & 2 \\ 4 & 5 \end{pmatrix} \quad S = \begin{pmatrix} 2 & 3 \\ 6 & 7 \end{pmatrix} \]

(1,2) joins with (2,3), but the other two tuples are dangling.

\[ R \text{ OUTERTJOIN} S \]

\[
\begin{array}{ccc}
A & B & C \\
1 & 2 & 3 \\
4 & 5 & \text{NULL} \\
\text{NULL} & 6 & 7 \\
\end{array}
\]
Outerjoins

Joins and Set operations in Relational Databases

- Inner join
- Left outer join
- Right outer join
- Full outer join
Outer-Joins

\[
\text{select [column list] from table_name}
\]
\[
\{\text{left | right | full}\} \text{ outer join table_name on qualification_list}
\]
\[
\text{where...}
\]

RA: $\mathcal{R}, \mathcal{N}, \mathcal{M}$
DATA DEFINITION LANGUAGE
Data Types in SQL

- **Character strings:**
  - CHAR(n): fixed-length string of n characters.
  - VARCHAR(n): string of length of up to n characters.

- **Bit strings:**
  - BIT(n): bit string of length n.
  - BIT VARYING(n): bit string of length upto n.

- **BOOLEAN:** possible values are TRUE, FALSE, and UNKNOWN

- **Integers:** INTEGER (INT), SHORTINT.

- **Floats:** FLOAT (or REAL), DOUBLE PRECISION.

- **Fixed point numbers:** DECIMAL(n, d): a number with n digits, with the decimal point d positions from the right.

- **Dates and times:** DATE and TIME

Read the Chapter!
Creating and Deleting Tables

- A table is a relation that is physically stored in a database.

- A table is **persistent**; it exists indefinitely unless dropped or altered in some way.

- Creating a table:
  - `CREATE TABLE` Students (PID VARCHAR(8), Name CHAR(20), Address VARCHAR(255));

- Deleting a table:
  - `DROP TABLE` followed by the name of the table.
Modifying Table Schemas

- **ALTER TABLE** followed by the name of the relation followed by:
  - **ADD** followed by a column name and its data type.
  - Add date of birth (Dob) to Students: `ALTER TABLE Students ADD Dob DATE;`
  - **DROP** followed by a column name.
Null Values

- null -> unknown, or inapplicable, (or ...)
- Complications:
  - 3-valued logic (true, false and unknown).
  - null = null : false!!
Null and Default Values

- SQL allows NULL for unknown attribute values.
- NULL not allowed in certain cases.

- We can specify a default value for an attribute using the DEFAULT keyword.
  - `ALTER TABLE Students ADD Gender char(1) DEFAULT '?';`
Inserting Data into a Table

- **INSERT INTO** `R(A1,A2, ... An) VALUES (v1, v2, ... , vn).
  - `(A1,A2, ... ,An)` can be a subset of `R`'s schema.
  - Remaining attributes get NULL values.
  - Can omit names of attributes if we provide values for all attributes and list values in standard order.

- Insertion: Instead of VALUES, can use a SELECT statement.
  - Insert into the Professors table all professors who are mentioned in Teach but are not in Professors.

  ```sql
  INSERT INTO Professors(PID)
  SELECT ProfessorPID
  FROM Teach
  WHERE ProfessorPID NOT IN
    (SELECT PID FROM Professors);
  ```
Deleting Data from a Table

- **DELETE FROM** R WHERE C.

- Every tuple satisfying the condition C is deleted from R.
Updating Data in a Table

- An update in SQL is a change to one of the tuples existing in the database.

- Example: change the name of a student so that every male student has ‘Mr.’ added to the name and every female student has ‘Ms.’ added to the name.
  
  - `UPDATE Students
    
    SET Name = 'Ms.' || Name
    
    WHERE Gender = 'F' ;`
  
  - `UPDATE Students
    
    SET Name = 'Mr.' || Name
    
    WHERE Gender = 'M' ;`

- Can set multiple attributes in the SET clause, separated by commas.

- The WHERE clause can involve a subquery.
Loading Data: BULK

- Different RDBMs have different syntax.
- PostgreSQL: Use the `\copy 'filename' INTO TABLE tablename;` at the `psql` prompt
- File format:
  - Tab-delimited with columns in the same order as the attributes.
  - Use `\N` to indicate null values.
- SQLite: see homework 1

- Do not make assumptions about how the RDBMS will behave!
- Check to make sure your data is not corrupted.
- Do not delete the original files that contain the raw data.
Saving Data

- Use the `pg_dump` program:
  - `pg_dump -t table database`

- Use `man pg_dump` for more information.
SQLite

- Most popular embedded db in the world
  - Iphone (iOS), Android, Chrome....
- (Very) Easy to use: no need to set it up
- Self-contained: data+schema
- DB on your laptop: useful for testing, understanding....
Views

- A view is a relation that does not exist physically.

- A view is defined by a query over other relations (tables and/or views).

- Just like a table, a view can be
  - queried: the query processor replaces the view by its definition.
  - used in other queries.

- Unlike a table, a view cannot be updated unless it satisfies certain conditions.
Example: View Definition

- CREATE VIEW ViewName AS Query;

- Suppose we want to perform a set of queries on those students who have taken courses both in the computer science and the mathematics departments.

- Let us create a view to store the PIDs of these students and the CS-Math course pairs they took.
Suppose we want to perform a set of queries on those students who have taken courses both in the computer science and the mathematics departments.

Let us create a view to store the PIDs of these students and the CS-Math course pairs they took.

```
CREATE VIEW CSMathStudents AS
  SELECT T1.StudentPID, T1.Number AS CSNum, T2.Number AS MathNum
  FROM Take AS T1, Take AS T2
  WHERE (T1.StudentPID = T2.StudentPID)
       AND (T1.DeptName = 'CS')
       AND (T2.DeptName = 'Math');
```
Querying Views

- Query a view as if it were a base table.
- How many students took both CS and Math courses?

SELECT COUNT(StudentPID)
FROM CSMathStudents
Querying Views

- Just replace view by its definition

```sql
SELECT COUNT(StudentPID)
FROM CSMathStudents
```

```sql
SELECT COUNT(StudentPID)
FROM
  (SELECT T1.StudentPID, T1.Number AS CSNum,
   T2.Number AS MathNum
  FROM Take AS T1, Take AS T2
  WHERE (T1.StudentPID = T2.StudentPID)
     AND (T1.DeptName = 'CS')
     AND (T2.DeptName = 'Math'));
```
Modifying Views

- What does it mean to modify a view?

- How is tuple deletion from a view executed?

- Can we insert a tuple into a view? Where will it be inserted, since a view does not physically exist?

- Can we insert tuples into any view? SQL includes rules that specify which views are updatable.
Deleting Views

- DROP VIEW CSMathStudents;

- Like a Symbolic Link: only the view definition is deleted
Deleting Tuples from Views

- Delete tuples for students taking 'CS 4604'.
  
  ```sql
  DELETE FROM CSMathStudents
  WHERE (CSNum = 4604);
  ```

- Deletion is executed as if were executing
  
  ```sql
  DELETE FROM Take
  WHERE (Number = 4604);
  ```

- Incorrect: non-CS tuples where (Number = 4604) will be deleted.
Deleting Tuples from Views

- Tuples only seen in the view should be deleted!
- Add conditions to the WHERE clause

```
DELETE FROM CSMathStudents
WHERE (CSNum = 4604) AND (DeptName = 'CS');
```
Inserting tuples into Views

- Again, passed through to the underlying relation

  \[
  \text{INSERT INTO CSMathStudents} \\
  \text{VALUES ('123-45-6789', 4604, 8811);} \\
  \]

- But Take schema is (PID, Number, Dept)
  - what should dept values be?
  - NULL?

Then it is not part of CSMathStudents!
Inserting tuples into Views

- CREATE VIEW CSStudents AS
  SELECT StudentPID, Number FROM Take
  WHERE (DeptName = 'CS');

- INSERT INTO CSStudents
  VALUES ('123-45-6789', 4604);
Inserting tuples into Views

- Include DeptName in the view's schema
- CREATE VIEW CSStudents AS
  SELECT StudentPID, DeptName, Number
  FROM Take
  WHERE (DeptName = 'CS');

- INSERT INTO CSStudents
  VALUES ('123-45-6789', 'CS', 4604)
Updatable Views

- The idea is that there must be a one-one relationship between rows in the view and the rows in the underlying table.
Updatable Views

SQL:92 standard:

- Defined by selecting/projecting some attributes from one relation R
- R may itself be an updatable view.
- Use SELECT and not SELECT DISTINCT.
- FROM clause can contain only one occurrence of R and must not contain any other relation.
- NO aggregation operations
Materialized Views

- Two kinds:
  1. **Virtual** = not stored in the database; just a query for constructing the relation.
  2. **Materialized** = actually constructed and stored.

WHY?
- Some views may be frequently used in queries.
- It may be efficient to materialize such a view, i.e., maintain its value at all times as a physical table.
Declaring Views

- Declare by:
  
  CREATE [MATERIALIZED] VIEW <name> AS <query>;

- Default is virtual.
Maintaining Materializing Views

- Cost?
  - Re-computing it when the underlying tables change
  - Materialized view may be much larger than original relations, e.g., in the case of joins
Maintaining Materialized Views

- CREATE MATERIALIZED VIEW CSStudents AS
  SELECT StudentPID, DeptName, Number
  FROM Take
  WHERE (DeptName = 'CS');

- When?
  - Insertion/deletion/update of Take

- Cost?
  - Insertion of tuple: Insert tuple into CSStudents only if new tuple has DeptName = 'CS'
  - Same for Deletion
  - Update? Delete followed by an Insert...
Maintaining Materialized Views

- Key idea is that many materialized views can be updated incrementally.

- Read Sections 25.9, and 25.10.1 from the textbook (~3 pages total)
Creating a Materialized View with Joins:

```
CREATE MATERIALIZED VIEW CSMathProfs(PID, Pname, CNum, CName) AS
  SELECT PID, P.Name, T.Number, T.Name
  FROM Teach AS T, Professors AS P
  WHERE (P.DeptName = 'CS') AND (T.DeptName = 'Math') AND
  (T.ProfessorPID = P.PID);
```

**Insert a tuple t into Teach:**

**Delete a tuple t from Teach:**
Maintaining Materialized Views with Joins

- CREATE MATERIALIZED VIEW CSMathProfs(PID, Pname, CNum, CName) AS
  SELECT PID, P.Name, T.Number, T.Name
  FROM Teach AS T, Professors AS P
  WHERE (P.DeptName = 'CS') AND (T.DeptName = 'Math') AND
  (T.ProfessorPID = P.PID);

- Insert a tuple t into Teach (assume t.DeptName = Math):
  Find the tuple p in Professors such that (t.ProfessorPID = p.PID) AND
  (p.DeptName = 'CS').
  Insert (p.PID, p.Name, t.Number, t.Name) into CSMathProfs

EXTRA: NOT IN EXAM
Maintaining Materialized Views with Joins

- CREATE MATERIALIZED VIEW CSMathProfs(PID, Pname, CNum, CName) AS
  SELECT PID, P.Name, T.Number, T.Name
  FROM Teach AS T, Professors AS P
  WHERE (P.DeptName = 'CS') AND (T.DeptName = 'Math') AND
  (T.ProfessorPID = P.PID);

- Delete a tuple t from Teach (assume t.DeptName = Math):
  DELETE FROM CSMathProfs WHERE CNum = t.Number;
Maintaining Materialized Views with Joins

- CREATE MATERIALIZED VIEW CSMathProfs(PID, Pname, CNum, CName) AS
  SELECT PID, P.Name, T.Number, T.Name
  FROM Teach AS T, Professors AS P
  WHERE (P.DeptName = 'CS') AND (T.DeptName = 'Math') AND
  (T.ProfessorPID = P.PID);

- Insert a tuple t into Professors:

- Delete a tuple t into Professors:
Maintaining Materialized Views with Joins

- CREATE MATERIALIZED VIEW CSMathProfs(PID, Pname, CNum, CName) AS
  
  SELECT PID, P.Name, T.Number, T.Name
  
  FROM Teach AS T, Professors AS P
  
  WHERE (P.DeptName = 'CS') AND (T.DeptName = 'Math') AND
  (T.ProfessorPID = P.PID);

- Insert a tuple t into Professors (assume p.DeptName = CS):
  
  INSERT INTO CSMathProfs
  
  SELECT p.PID, p.Name, T.Number, T.Name
  
  WHERE (p.PID = T.ProfessorPID) AND (T.DeptName = 'Math');
Maintaining Materialized Views with Joins

- CREATE MATERIALIZED VIEW CSMathProfs(PID, Pname, CNum, CName) AS
  SELECT PID, P.Name, T.Number, T.Name
  FROM Teach AS T, Professors AS P
  WHERE (P.DeptName = 'CS') AND (T.DeptName = 'Math') AND
  (T.ProfessorPID = P.PID);

- Delete a tuple t from Professors (assume p.DeptName = CS):
  DELETE FROM CSMathProfs WHERE (PID = p.PID);
Periodic Maintenance

- DB for inventory of a department store.
- Aggregate buyer patterns for further analysis → can be a (materialized) view
- Analysis is only periodic, so update the materialized view at only regular intervals

EXTRA: NOT IN EXAM
In practice, views are materialized because they are helpful to answer common queries.

Can we rewrite a query to use a materialized view rather than the original relations?

EXTRA: NOT IN EXAM
Rewriting Queries Using Materialized Views

- Find names and addresses of students taking CS courses
  
  ```sql
  SELECT Name, Address
  FROM Students, Take
  WHERE (Students.PID = Take.StudentPID) AND
  (DeptName = 'CS');
  ```

Rewrite it using CSStudents?

```sql
SELECT Name, Address
FROM Students, CSStudents
WHERE (Students.PID = CSStudents.StudentPID);
```
Rules for Rewriting Queries

- Complete sets of rules is very complex!
- A simple rule

**View V:**

```
SELECT LV
FROM RV
WHERE CV
```

**Query Q:**

```
SELECT LQ
FROM RQ
WHERE CQ
```

**(New) Query Q’:**

```
SELECT LQ
FROM V, RQ - RV
WHERE C
```

- We can replace Q by the new query Q’ if
  - \( RV \subseteq RQ \)
  - \( CQ \equiv CV \text{ AND } C \), for some condition \( C \), which may be empty
  - If \( C \) is not empty, then attributes of relations in \( RV \) that \( C \) mentions are also in \( LV \)
  - Attributes in \( LQ \) that come from relations in \( RV \) are also in the list of attributes \( LV \)