Carnegie Mellon Univ.  
Dept. of Computer Science  
15-415/615 - DB Applications  

C. Faloutsos – A. Pavlo  
Lecture#7: Fun with SQL (Part 2)

Last Class
- Basic Queries
- Table Definition (DDL)
- NULLs
- String/Date/Time/Set/Bag Operations
- Output Redirection/Control

Today's Class: OLAP
- Views
- Joins
- Aggregations + Group By
- Nested Queries
- Window Functions
- Common Table Expressions

Example Database

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Kayne</td>
<td>kayne@cs</td>
<td>39</td>
<td>4.0</td>
</tr>
<tr>
<td>53688</td>
<td>Bieber</td>
<td>jbieber@cs</td>
<td>22</td>
<td>3.9</td>
</tr>
<tr>
<td>53655</td>
<td>Tupac</td>
<td>shakur@cs</td>
<td>26</td>
<td>3.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-415</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53655</td>
<td>15-415</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-415</td>
<td>Database Applications</td>
</tr>
<tr>
<td>15-721</td>
<td>Database Systems</td>
</tr>
<tr>
<td>15-826</td>
<td>Data Mining</td>
</tr>
<tr>
<td>15-823</td>
<td>Advanced Topics in Databases</td>
</tr>
</tbody>
</table>
Views

• Creates a “virtual” table containing the output from a **SELECT** query.
• Mechanism for hiding data from view of certain users.
• Can be used to simplify a complex query that is executed often.
  – Won’t make it faster though!

Faloutsos/Pavlo
CMU SCS 15-415/615 5

View Example

• Create a view of the CS student records with just their id, name, and login.

```sql
CREATE VIEW CompSciStudentInfo AS
  SELECT sid, name, login
  FROM student
  WHERE login LIKE '%@cs';
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Kayne</td>
<td>kayne@cs</td>
<td>45</td>
<td>4.0</td>
</tr>
<tr>
<td>53688</td>
<td>Bieber</td>
<td>jbieber@cs</td>
<td>21</td>
<td>3.9</td>
</tr>
</tbody>
</table>

Original Table

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Kayne</td>
<td>kayne@cs</td>
<td>45</td>
<td>4.0</td>
</tr>
<tr>
<td>53688</td>
<td>Bieber</td>
<td>jbieber@cs</td>
<td>21</td>
<td>3.9</td>
</tr>
</tbody>
</table>

View

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Kayne</td>
<td>kayne@cs</td>
</tr>
<tr>
<td>53688</td>
<td>Bieber</td>
<td>jbieber@cs</td>
</tr>
</tbody>
</table>

View Example

• Create a view with the average age of the students enrolled in each course.

```sql
CREATE VIEW CourseAge AS
  SELECT cid, AVG(age) AS avg_age
  FROM student, enrolled
  WHERE student.sid = enrolled.sid
  GROUP BY enrolled.cid;
```

<table>
<thead>
<tr>
<th>cid</th>
<th>avg_age</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-415</td>
<td>45.0</td>
</tr>
<tr>
<td>15-721</td>
<td>45.0</td>
</tr>
<tr>
<td>15-826</td>
<td>21.0</td>
</tr>
</tbody>
</table>

View Example

• Create a view with the average age of the students enrolled in each course.

```sql
CREATE VIEW CourseAge AS
  SELECT cid, AVG(age) AS avg_age
  FROM student, enrolled
  WHERE student.sid = enrolled.sid
  GROUP BY enrolled.cid;
```

<table>
<thead>
<tr>
<th>cid</th>
<th>avg_age</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-415</td>
<td>45.0</td>
</tr>
<tr>
<td>15-721</td>
<td>45.0</td>
</tr>
<tr>
<td>15-826</td>
<td>21.0</td>
</tr>
</tbody>
</table>
**Views vs. SELECT INTO**

```sql
CREATE VIEW AvgGPA AS
    SELECT AVG(gpa) AS avg_gpa FROM student
    WHERE login LIKE '%@cs'
```

```sql
SELECT AVG(gpa) AS avg_gpa INTO AvgGPA
    FROM student WHERE login LIKE '%@cs'
```

- **INTO** → Creates static table that does not get updated when student gets updated.
- **VIEW** → Dynamic results are only materialized when needed.

**Materialized Views**

- Creates a view containing the output from a `SELECT` query that is automatically updated when the underlying tables change.

```sql
CREATE MATERIALIZED VIEW AvgGPA AS
    SELECT AVG(gpa) AS avg_gpa FROM student
    WHERE login LIKE '%@cs'
```

**Today's Class: OLAP**

- Views
- Joins
- Aggregations + Group By
- Nested Queries
- Window Functions
- Common Table Expressions
Join Query Grammar

- **Join-Type**: The type of join to compute.
- **Qualification**: Expression that determines whether a tuple from table1 can be joined with table2. Comparison of attributes or constants using operators =, ≠, <, >, ≤, and ≥.

### INNER JOIN

```
SELECT name, cid, grade
FROM student, enrolled
WHERE student.sid = enrolled.sid
```

### OUTER JOIN

**LEFT OUTER JOIN**

```
SELECT name, cid, grade
FROM student
LEFT OUTER JOIN enrolled
ON student.sid = enrolled.sid
```

**RIGHT OUTER JOIN**

```
SELECT name, cid, grade
FROM enrolled
RIGHT OUTER JOIN student
ON student.sid = enrolled.sid
```

**NATURAL JOIN**

```
SELECT name, cid, grade
FROM student
NATURAL JOIN enrolled
```
Join Types

\[
\text{SELECT * FROM A JOIN B ON A.id = B.id}
\]

<table>
<thead>
<tr>
<th>Join Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INNER JOIN</td>
<td>Join where A and B have same value</td>
</tr>
<tr>
<td>LEFT OUTER JOIN</td>
<td>Join where A and B have same value AND where only A has a value</td>
</tr>
<tr>
<td>RIGHT OUTER JOIN</td>
<td>Join where A and B have same value AND where only B has a value</td>
</tr>
<tr>
<td>FULL OUTER JOIN</td>
<td>Join where A and B have same value AND where A or B have unique values</td>
</tr>
<tr>
<td>CROSS JOIN</td>
<td>Cartesian Product</td>
</tr>
</tbody>
</table>

Today's Class: OLAP

- Views
- Joins
- Aggregations + Group By
- Nested Queries
- Window Functions
- Common Table Expressions

Aggregates

- Functions that return a single value from a bag of tuples:
  - \( \text{AVG(col)} \) → Return the average col value.
  - \( \text{MIN(col)} \) → Return minimum col value.
  - \( \text{MAX(col)} \) → Return maximum col value.
  - \( \text{SUM(col)} \) → Return sum of values in col.
  - \( \text{COUNT(col)} \) → Return # of values for col.

Aggregates

- Functions can only be used in the \text{SELECT} attribute output list.
- \textit{Get the # of students with a “@cs” login:}
  \[
  \text{SELECT COUNT(login) AS cnt FROM student WHERE login LIKE ‘%@cs’}
  \]
Aggregates

- Functions can only be used in the **SELECT** attribute output list.
- *Get the # of students with a “@cs” login:*

```
SELECT COUNT(login) AS cnt
FROM student WHERE login LIKE '%@cs'
```

- **cnt**
  - 12

- Can use multiple functions together at the same time.
  - *Get the number of students and their average GPA that have a “@cs” login.***

```
SELECT AVG(gpa), COUNT(sid)
FROM student WHERE login LIKE '%@cs'
```

- **AVG(gpa)  COUNT(sid)**
  - 3.25  12
Aggregates

• **COUNT, SUM, AVG** support **DISTINCT**

• Get the number of unique students that have an “@cs” login.

```sql
SELECT COUNT(DISTINCT login)
FROM student WHERE login LIKE '%@cs'
```

• Output of other columns outside of an aggregate is undefined:

```sql
SELECT AVG(s.gpa), e.cid
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
```

• Unless...

```sql
SELECT COUNT(DISTINCT login)
FROM student WHERE login LIKE '%@cs'
```

```
<table>
<thead>
<tr>
<th>COUNT(DISTINCT login)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
</tbody>
</table>
```

```sql
SELECT AVG(s.gpa), e.cid
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
```

```
<table>
<thead>
<tr>
<th>AVG(s.gpa)</th>
<th>e.cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>???</td>
</tr>
</tbody>
</table>
```
GROUP BY

- Project tuples into subsets and calc aggregates against each subset.

```
SELECT AVG(s.gpa), e.cid
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
GROUP BY e.cid
```

GROUP BY

- Project tuples into subsets and calc aggregates against each subset.

```
SELECT AVG(s.gpa), e.cid
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
GROUP BY e.cid
```

GROUP BY

- Non-aggregated values in SELECT output clause must appear in GROUP BY clause.

```
SELECT AVG(s.gpa), e.cid, s.name
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
GROUP BY e.cid
```

GROUP BY

- Non-aggregated values in SELECT output clause must appear in GROUP BY clause.

```
SELECT AVG(s.gpa), e.cid, s.name
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
GROUP BY e.cid, s.name
```
**HAVING**

- Filters output results
- Like a `WHERE` clause for a `GROUP BY`

```sql
SELECT AVG(s.gpa) AS avg_gpa, e.cid
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
 GROUP BY e.cid
 HAVING avg_gpa > 2.75;
```

<table>
<thead>
<tr>
<th>AVG(s.gpa)</th>
<th>e.cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.46</td>
<td>15-721</td>
</tr>
<tr>
<td>3.39</td>
<td>15-826</td>
</tr>
<tr>
<td>1.89</td>
<td>15-415</td>
</tr>
</tbody>
</table>

Today's Class: OLAP

- Views
- Joins
- Aggregations + Group By
- Nested Queries
- Window Functions
- Common Table Expressions
Nested Queries

• Queries containing other queries
  “outer query” “inner query”
  “inner query:
  – Can appear in FROM or WHERE clause

SELECT name FROM student WHERE sid IN (SELECT sid FROM enrolled)

Think of this as a function that returns the result of the inner query

• Find the names of students in ‘15-415’

SELECT name FROM student
WHERE ...
  sid in the set of people that take 15-415

Faloutsos/Pavlo

CMU SCS 15-415/615

28

Nested Queries

• Find the names of students in ‘15-415’

SELECT name FROM student
WHERE ...
  SELECT sid FROM enrolled
  WHERE cid = ‘15-415’

Faloutsos/Pavlo

CMU SCS 15-415/615

29

Nested Queries

• Find the names of students in ‘15-415’

SELECT name FROM student
WHERE ...
  SELECT sid FROM enrolled
  WHERE cid = ‘15-415’

Faloutsos/Pavlo

CMU SCS 15-415/615

30
Nested Queries

• **ALL** → Must satisfy expression for all rows in sub-query
• **ANY** → Must satisfy expression for at least one row in sub-query.
• **IN** → Equivalent to ‘=ANY()’.
• **EXISTS** → At least one row is returned.

• Nested queries are difficult to optimize. Try to avoid them if possible.

Find the names of students in ‘15-415’

```sql
SELECT name FROM student WHERE sid = ANY(
   SELECT sid FROM enrolled
   WHERE cid = '15-415'
)
```

Find student record with the highest id.

• This won’t work in **SQL-92**:

```sql
SELECT MAX(sid), name FROM student;
```

• Runs in **MySQL**, but you get a wrong answer:

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>53688</td>
<td>Tupac</td>
</tr>
</tbody>
</table>

“is greater than every other sid”
Nested Queries

• Find student record with the highest id.

```
SELECT sid, name FROM student
WHERE sid => ALL(
  SELECT sid FROM enrolled
)
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>53688</td>
<td>Bieber</td>
</tr>
</tbody>
</table>

• Find all courses that nobody is enrolled in.

```
SELECT * FROM course
WHERE ...
```

“with no tuples in the ‘enrolled’ table”

<table>
<thead>
<tr>
<th>cid</th>
<th>name</th>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-415</td>
<td>Database Applications</td>
<td>53666</td>
<td>15-415</td>
<td>C</td>
</tr>
<tr>
<td>15-721</td>
<td>Database Systems</td>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>15-826</td>
<td>Data Mining</td>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>15-823</td>
<td>Advanced Topics in Databases</td>
<td>53655</td>
<td>15-415</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td></td>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
</tbody>
</table>
Nested Queries

• Find all courses that nobody is enrolled in.

```sql
SELECT * FROM course
WHERE NOT EXISTS(
    SELECT * FROM enrolled
    WHERE course.cid = enrolled.cid
)
```

<table>
<thead>
<tr>
<th>cid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-823</td>
<td>Advanced Topics in Databases</td>
</tr>
</tbody>
</table>

Today's Class: OLAP

• Views
• Joins
• Aggregations + Group By
• Nested Queries
• Window Functions
• Common Table Expressions
**Window Functions**

- Performs a calculation across a set of tuples that related to a single row.
- Like an aggregation but tuples are not grouped into a single output tuples.

```
SELECT ... FUNC-NAME(...) OVER (...) 
FROM tableName
```

**Aggregation Functions**
- Anything that we discussed earlier

**Special Functions**
- `ROW_NUMBER()` → Number of the current row
- `RANK()` → Order position of the current row.

```
SELECT *, ROW_NUMBER() OVER () 
FROM enrolled
```

**Window Function**

- The **OVER** keyword specifies how to group together tuples when computing the window function.
- Use **PARTITION BY** to specify group.

```
SELECT *
    ROW_NUMBER() OVER (PARTITION BY cid) 
FROM enrolled
ORDER BY cid
```

**Window Function**

- You can also include an **ORDER BY** in the window grouping.

```
SELECT *
    ROW_NUMBER() OVER (ORDER BY cid) 
FROM enrolled
ORDER BY cid
```
Window Functions

- Find the student with the highest grade for each course.

\[
\text{SELECT } *, \text{ RANK() \text{OVER} (PARTITION BY \text{cid} \text{ORDER BY grade ASC})} \\
\text{FROM enrolled}
\]

Window Functions

- Get the name of the student with the second highest grade for each course.

\[
\text{SELECT } * \text{ FROM (} \\
\text{SELECT C.name, S.name, E.grade,} \\
\text{RANK() \text{OVER} (PARTITION BY E.cid \text{ORDER BY E.grade ASC})} \\
\text{) AS grade_rank} \\
\text{FROM student S, course C, enrolled E} \\
\text{WHERE S.sid = E.sid} \\
\text{AND C.cid = E.cid} \\
\text{) AS R} \\
\text{WHERE R.grade_rank = 2;}
\]

Today's Class: OLAP

- Views
- Joins
- Aggregations + Group By
- Nested Queries
- Window Functions
- Common Table Expressions

Common Table Expressions

- Provides a way to write auxiliary statements for use in a larger query.
  - Think of it like a temp table just for one query.
- Alternative to nested queries and views.

\[
\text{WITH cteName AS (} \\
\text{SELECT 1} \\
\text{)} \\
\text{SELECT } * \text{ FROM cteName}
\]
Common Table Expressions

- You can bind output columns to names before the **AS** keyword.

```sql
WITH cteName (col1, col2) AS (
    SELECT 1, 2
)  
SELECT col1 + col2 FROM cteName
```

- **Find student record with the highest id that is enrolled in at least one course.**

```sql
WITH cteSource (maxId) AS (
    SELECT MAX(sid) FROM enrolled
)  
SELECT name FROM student, cteSource  
WHERE student.sid = cteSource.maxId
```

CTEs – Recursion

- **Print 1 to 10.**

```sql
WITH RECURSIVE cteSource (counter) AS (  
    (SELECT 1)  
    UNION ALL  
    (SELECT counter + 1 FROM cteSource  
    WHERE counter < 10)  
)  
SELECT * FROM cteSource
```

- **Postgres CTE Demo!**

Next Class

- We begin discussing storage internals.
- This material will be important for helping you pick up dates at parties.