A SQL query walks up to two tables in a restaurant and asks: “Mind if I join you?”

SQL:
Structured Query Language
Data Manipulation Language (DML)

CSC343, Introduction to Databases
Nosayba El-Sayed (based on slides from Diane Horton)
Fall 2015
Few things from last week (RA)

16. Department and cNum of all courses that have been taught in every term when csc448 was taught.

Answer:

$448Terms(\text{term}) := \Pi_{\text{term}}(\sigma_{\text{dept}=\text{“csc”}} \land \text{cNum}=448\ Offer)\times\Pi_{\text{dept, cNum} \text{Course} \times 448Terms}\\
\text{ShouldHaveBeen}(\text{dept, cNum, term}) := \Pi_{\text{dept, cNum} \text{Course} \times 448Terms}\\
\text{CourseTerms}(\text{dept, cNum, term}) := \Pi_{\text{dept, cNum, term} \text{Offering}}\\
\text{WereNotAlways}(\text{dept, cNum, term}) := \text{ShouldHaveBeen} - \text{CourseTerms}\\
\text{Answer}(\text{dept, cNum}) := (\Pi_{\text{dept, cNum} \text{Course}}) - (\Pi_{\text{dept, cNum} \text{WereNotAlways}})\times$
Evaluating Queries:

- Any problem has multiple RA solutions.
  - Each solution suggests a “query execution plan”.
  - Some may seem more efficient.
- But in RA, we won’t care about efficiency; it’s an algebra.

- In a DBMS, queries actually are executed, & efficiency matters!
  - Which query execution plan is most efficient depends on the data in the database and what indices you have.
  - Fortunately, the DBMS optimizes our queries.
  - We can focus on what we want, not how to get it.
Speaking of which..

• You still want to design your SQL queries carefully for them to run efficiently!

• Example: PCRS Week 4 Prep
  • Some students joined all the tables in the schema; DBMS wasn’t happy about it.
Not related to class..

Elections Canada office on campus for early voting

OCTOBER 5th to 8th – DAILY 10am to 8pm
Elections Canada
316 Bloor St W.

UTGSU – GYM
16 Bancroft Ave

Walfond Centre
36 Harbord St

More info: elections.ca

Canadian Voter Turnout

Ages [18-24]: 39%

~60%
SQL -- Introduction

• So far, we have defined database schemas and queries mathematically.

• SQL is a formal language for doing so with a DBMS.

• “Structured Query Language”, but it’s for more than writing queries.

• Two sub-parts:
  • DDL (Data Definition Language), for defining schemas.
  • DML (Data Manipulation Language), for writing queries and modifying the database.
PostgreSQL

• We’ll be working in PostgreSQL, an open-source relational DBMS.
• Learn your way around the documentation; it will be very helpful.
• Standards?
  • There are several, the most recent being SQL:2008.
  • PostgreSQL supports most of it SQL:2008.
  • DBMSs vary in the details around the edges, making portability difficult.
Why the elephant?

Re: [HACKERS] PostgreSQL logo.

Author: yang(at)sjphil(dot)sju(dot)edu
Date: 1997-04-03 20:36:33
Subject: Re: [HACKERS] PostgreSQL logo.

Some other ideas:

derivative: a sword (derivative of the Dragon book cover)
illustrative: a bowl of Alphabet Soup, with letters spelling out PostgreSQL
obscure: a revolver/hit man (Grosse Pt is an anagram of Postgres, and an abbreviation of the title of the new John Cusack movie)

but if you want an animal-based logo, how about some sort of elephant? After all, as the Agatha Christie title read, elephants can remember ...

David Yang
A high-level language

• SQL is a very high-level, *declarative* language.
  • Say “what” rather than “how.”
  • Contrast to languages like Java or C++ (*imperative*).

• Provides physical “*data independence*”
  • Details of how the data is stored can change with no impact on your queries!

• You can focus on *readability*.
  • But because the DMBS optimizes your query, you get *efficiency*. 
Heads up: **SELECT vs σ**

- **In SQL,**
  - **SELECT** is for choosing *columns*, i.e., $\pi$.
  - Example:
    ```sql
    select surName
    from Student
    where campus = 'StG';
    ```

- **In relational algebra,**
  - “select” means choosing *rows*, i.e., $\sigma$.
Basic queries

```
SELECT attributes
FROM Table
WHERE <condition>;
```
Meaning of a query with one relation

```sql
SELECT name
FROM Course
WHERE dept = 'CSC';
```

\[ \pi_{\text{name}} (\sigma_{\text{dept} = \text{"csc"}} (\text{Course})) \]
... and with multiple relations

```sql
SELECT name 
FROM Course, Offering, Took 
WHERE dept = 'CSC';
```

\[
\pi_{\text{name}} (\sigma_{\text{dept}='csc'} (\text{Course} \times \text{Offering} \times \text{Took}))
\]
Temporarily renaming a table

• You can rename tables (just for the duration of the statement):

```
SELECT e.name, d.name
FROM employee e, department d
WHERE d.name = 'marketing'
    and e.name = 'Horton';
```

• This is like $\rho$ in relational algebra.

• Can be convenient vs the longer full names:

```
SELECT employee.name, department.name
FROM employee, department
WHERE department.name = 'marketing'
    and employee.name = 'Horton';
```
Self-joins

- As we know, renaming is **required** for self-joins.
- Example:
  
  ```sql
  select e1.name, e2.name
  from employee e1, employee e2
  where e1.salary < e2.salary;
  ```
Using * In SELECT clauses

• A * in the SELECT clause means “all attributes of this relation.”

• Example:

```
SELECT *
FROM Course
WHERE dept = 'CSC';
```
Renaming attributes

• Use `AS «new name»` to rename an attribute in the result.

• Example:
  
  ```sql
  SELECT name AS title, dept
  FROM Course
  WHERE breadth;
  ```
Complex Conditions in a WHERE

• We can build boolean expressions with operators that produce boolean results.
  • comparison operators: =, <>, <, >, <=, >=
  • and many other operators: see section 6.1.2 of the text and chapter 9 of the PostgreSQL documentation.

• We can combine boolean expressions with:
  • Boolean operators: AND, OR, NOT.
Example: Compound condition

• Find 3rd- and 4th-year CSC courses:

```
SELECT *  
FROM Offering  
WHERE dept = 'CSC' AND cnum >= 300;
```
ORDER BY

• To put the tuples in order, add this as the final clause:
  ORDER BY «attribute list» [DESC]
• The default is ascending order; DESC overrides it to force descending order.
• The attribute list can include expressions: e.g.,
  ORDER BY sales+rentals
• The ordering is the last thing done before the SELECT, so all attributes are still available.
Case-sensitivity and whitespace

• Example query:

```sql
select surName
from Student
where campus = 'StG';
```

• Keywords, like `select`, are not case-sensitive.
  • One convention is to use `UPPERCASE` for keywords.

• Identifiers, like `Student` are not case-sensitive either.
  • One convention is to use lowercase for attributes, and a leading capital letter followed by lowercase for relations.

• Literal strings, like `'StG'`, are case-sensitive, and require single quotes.

• Whitespace (other than inside quotes) is ignored.
Expressions in SELECT clauses

• Instead of a simple attribute name, you can use an expression in a SELECT clause.

• Operands: attributes, constants
  Operators: arithmetic ops, string ops

• Examples:
  SELECT sid, grade-10 as adjusted
  FROM Took;

  SELECT dept||cnum
  FROM course;

  --Note that || is string concatenation
Expressions that are a constant

• Sometimes it makes sense for the whole expression to be a constant (something that doesn’t involve any attributes!).

• Example:

```
SELECT name, 'satisfies' AS breadthRequirement
FROM Course
WHERE breadth;
```
Pattern operators

- **Two ways to compare a string to a pattern by:**
  - «attribute» LIKE «pattern»
  - «attribute» NOT LIKE «pattern»

- **Pattern is a quoted string**
  - % means: any string
  - _ means: any single character

- **Example:**

  ```sql
  SELECT *
  FROM Course
  WHERE name LIKE '%%Comp%%';
  ```
Pattern operators – More Examples

• ... WHERE phone LIKE ‘268_ _ _ _ _’
  • phone numbers with area code 268

• ... WHERE Dictionary.entry NOT LIKE ‘%est’
  • Ignore ‘biggest’, ‘tallest’, ‘fastest’, ‘rest’, ...

• ... WHERE sales LIKE '30!%%' ESCAPE ‘!’
  • How about: Sales of 30%?
Aggregation
Computing on a column

• We often want to compute something across the values in a column.
• \texttt{SUM}, \texttt{AVG}, \texttt{COUNT}, \texttt{MIN}, and \texttt{MAX} can be applied to a column in a SELECT clause.
• Also, \texttt{COUNT(*)} counts the number of tuples.
• We call this aggregation.
• Note: To stop duplicates from contributing to the aggregation, use \texttt{DISTINCT} inside the brackets.
Computing on a column - Examples

- SELECT * FROM TOOK;

<table>
<thead>
<tr>
<th>sid</th>
<th>oid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>1</td>
<td>65</td>
</tr>
<tr>
<td>12345</td>
<td>3</td>
<td>38</td>
</tr>
<tr>
<td>12345</td>
<td>4</td>
<td>82</td>
</tr>
<tr>
<td>99999</td>
<td>3</td>
<td>80</td>
</tr>
<tr>
<td>21111</td>
<td>3</td>
<td>87</td>
</tr>
<tr>
<td>21111</td>
<td>1</td>
<td>58</td>
</tr>
<tr>
<td>41111</td>
<td>1</td>
<td>82</td>
</tr>
<tr>
<td>31111</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>31111</td>
<td>3</td>
<td>77</td>
</tr>
<tr>
<td>31111</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>31111</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>55555</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>55555</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>55555</td>
<td>5</td>
<td>79</td>
</tr>
<tr>
<td>55555</td>
<td>4</td>
<td>88</td>
</tr>
</tbody>
</table>

(15 rows)
Computing on a column - Examples

- **SELECT grade FROM TOOK;**

Yes.. Duplicates are *not* eliminated by default when using SELECT…

<table>
<thead>
<tr>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>82</td>
</tr>
<tr>
<td>80</td>
</tr>
<tr>
<td>87</td>
</tr>
<tr>
<td>58</td>
</tr>
<tr>
<td>82</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>77</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>100</td>
</tr>
<tr>
<td>79</td>
</tr>
<tr>
<td>88</td>
</tr>
<tr>
<td>(15 rows)</td>
</tr>
</tbody>
</table>
Computing on a column - Examples

- SELECT $\text{AVG}(\text{grade})$ FROM TOOK;

- SELECT $\text{AVG}(\text{grade})$ as myAvg FROM TOOK;
Computing on a column - Examples

- **SELECT** `max(grade), avg(grade), count(*), min(sid)`
  
  **FROM TOOK**

<table>
<thead>
<tr>
<th>max</th>
<th>avg</th>
<th>count</th>
<th>min</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>82.4000000000000000</td>
<td>15</td>
<td>12345</td>
</tr>
</tbody>
</table>

(1 row)
Computing on columns

• Now what if we want to compute aggregates (e.g.: AVG grade) for each offering separately?

Took($sID$, $oID$, grade)

Avg1 grades for $oID$=1 ...
Avg2 grades for $oID$=2
Avg3 grades for $oID$=3
....etc
Grouping-By

• If we follow a SELECT-FROM-WHERE expression with `GROUP BY <attributes>`
  • The rows are grouped together according to the values of those attributes, and
  • any aggregation is applied only within each group.
Grouping-By Example

- SELECT oid, avg(grade) as offavg
  FROM took
  GROUP BY oid;

Note that your SELECT can’t include un-aggregated columns (e.g. sid).

What if we want to know the number of students in each offering?
Grouping-By Example

- SELECT oID, avg(grade) as offavg, count(*) as numstudents
  FROM took
  GROUP BY oID;

<table>
<thead>
<tr>
<th>oid</th>
<th>offavg</th>
<th>numstudents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>81.00000</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>70.50000</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>89.50000</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>85.00000</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>100.0000</td>
<td>2</td>
</tr>
</tbody>
</table>

Took

<table>
<thead>
<tr>
<th>sid</th>
<th>oid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>1</td>
<td>65</td>
</tr>
<tr>
<td>12345</td>
<td>3</td>
<td>38</td>
</tr>
<tr>
<td>12345</td>
<td>4</td>
<td>82</td>
</tr>
<tr>
<td>99999</td>
<td>3</td>
<td>80</td>
</tr>
<tr>
<td>21111</td>
<td>3</td>
<td>87</td>
</tr>
<tr>
<td>21111</td>
<td>1</td>
<td>58</td>
</tr>
<tr>
<td>41111</td>
<td>1</td>
<td>82</td>
</tr>
<tr>
<td>31111</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>31111</td>
<td>3</td>
<td>77</td>
</tr>
<tr>
<td>31111</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>31111</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>55555</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>55555</td>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>55555</td>
<td>5</td>
<td>79</td>
</tr>
<tr>
<td>55555</td>
<td>4</td>
<td>88</td>
</tr>
</tbody>
</table>

(15 rows)
Restrictions on aggregation

• If any aggregation is used, then each element of the SELECT list must be either:
  • aggregated, or
  • an attribute on the GROUP BY list.

• Otherwise, it doesn’t even make sense to include the attribute.
Grouping-By Example

- SELECT `oID`, avg(grade) as offavg
  FROM took GROUP BY `oID`;

Sometimes we want to keep some groups and eliminate others from our result set.

What if we want to learn which offerings had an average > 80 only?

What if we want to get the avg grade for offerings with oID < 5 ...

<table>
<thead>
<tr>
<th>oID</th>
<th>offavg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>81.00000</td>
</tr>
<tr>
<td>3</td>
<td>70.50000</td>
</tr>
<tr>
<td>5</td>
<td>89.50000</td>
</tr>
<tr>
<td>4</td>
<td>85.00000</td>
</tr>
<tr>
<td>6</td>
<td>100.00000</td>
</tr>
</tbody>
</table>
HAVING Clauses

- WHERE let’s you decide which tuples to keep.
- Similarly, you can decide which groups to keep.
- Syntax:
  
  ```
  ... 
  GROUP BY «attributes»
  HAVING «condition»
  ```
- Semantics:
  Only groups satisfying the condition are kept.
Requirements on HAVING clauses

• Outside subqueries, HAVING may refer to attributes only if they are either:
  • aggregated, or
  • an attribute on the GROUP BY list.
• (The same requirement as for SELECT clauses with aggregation).
HAVING Examples

• SELECT oID, avg(grade) as offavg
  FROM took
  GROUP BY oID
  HAVING avg(grade)>80;

<table>
<thead>
<tr>
<th>oid</th>
<th>offavg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>81.00000</td>
</tr>
<tr>
<td>5</td>
<td>89.50000</td>
</tr>
<tr>
<td>4</td>
<td>85.00000</td>
</tr>
<tr>
<td>6</td>
<td>100.00000</td>
</tr>
</tbody>
</table>

• SELECT oID, avg(grade) as offavg
  FROM took
  GROUP BY oID
  HAVING oID <= 5
  ORDER BY oID;

<table>
<thead>
<tr>
<th>oid</th>
<th>offavg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>81.00000</td>
</tr>
<tr>
<td>3</td>
<td>70.50000</td>
</tr>
<tr>
<td>4</td>
<td>85.00000</td>
</tr>
<tr>
<td>5</td>
<td>89.50000</td>
</tr>
</tbody>
</table>
-- Class Exercise Time --
Basic SQL, Aggregates
1. Write a query to find the AVG, MIN, and MAX grade for each Offering:

```sql
SELECT oID, AVG(grade), MIN(grade), MAX(grade)
FROM Took
GROUP BY oID
```

**ROSI Schema**

- Students(sID, surName, campus)
- Courses(dept, cNum, cName, br)
- Offerings(oID, dept, cNum, term, inst)
- Took(sID, oID, grade)
3. Find the sID and avg grade of each student, but keep data only for students with sID > 22222:

\[
\text{SELECT } \ sID, \ \text{AVG(grade)} \\
\text{FROM } \ \text{Took} \\
\text{WHERE } \ sID > 22222 \\
\text{HAVING } \ sID > 22222 \\
\text{ORDER BY } \\
\]

**ROSI Schema**

- Students(sID, surName, campus)
- Courses(dept, cNum, cName, br)
- Offerings(oID, dept, cNum, term, inst)
- Took(sID, oID, grade)
3. Find the sID and avg grade of each student, but keep data only for students with sID > 22222:

```
SELECT  sID, AVG(grade) 
FROM    Took 
WHERE   sID > 22222 
GROUP BY sID 
HAVING  
ORDER BY 
```

**ROSI Schema**

- Students(sID, surName, campus)
- Courses(dept, cNum, cName, br)
- Offerings(oID, dept, cNum, term, inst)
- Took(sID, oID, grade)
4. Find only the sID of each student with an average over 80:

```
SELECT sID, AVG(grade) as studentAvg
FROM Took
GROUP BY sID
HAVING AVG(grade) > 80
ORDER BY
```

Note: Can’t use the alias `studentAvg` here…

ROSIS Schema

- Students(sID, surName, campus)
- Courses(dept, cNum, cName, br)
- Offerings(oID, dept, cNum, term, inst)
- Took(sID, oID, grade)
5. Which of these queries is legal?

- SELECT dept
  FROM Took, Offering
  WHERE Took.oID = Offering.oID
  GROUP BY dept
  HAVING avg(grade) > 75;

- SELECT Took.oID, dept, cNum, avg(grade)
  FROM Took, Offering
  WHERE Took.oID = Offering.oID
  GROUP BY Took.oID
  HAVING avg(grade) > 75;

- SELECT Took.oID, avg(grade)
  FROM Took, Offering
  WHERE Took.oID = Offering.oID
  GROUP BY Took.oID
  HAVING avg(grade) > 75;

- SELECT oID, avg(grade)
  FROM Took
  GROUP BY sID
  HAVING avg(grade) > 75;

---

**ROSI Schema**

Students(sID, surName, campus)
Courses(dept, cNum, cName, br)
Offerings(oID, dept, cNum, term, inst)
Took(sID, oID, grade)
Set operations
Tables can have duplicates in SQL

• A table can have duplicate tuples, unless this would violate an integrity constraint.
• And SELECT-FROM-WHERE statements leave duplicates in unless you say not to.

• Why?
  • Getting rid of duplicates is expensive! $$$
  • We may want the duplicates because they tell us how many times something occurred.
Bags

• SQL treats tables as “bags” (or “multisets”) rather than sets.
• Bags are just like sets, but duplicates are allowed.
• \{6, 2, 7, 1, 9\} is a set (and a bag)
  \{6, 2, 2, 7, 1, 9\} is not a set, but is a bag.
• Like with sets, order doesn’t matter.
  \{6, 2, 7, 1, 9\} = \{1, 2, 6, 7, 9\}
Union, Intersection, and Difference

- These are expressed as:
  
  \[
  (\text{«subquery»}) \ \text{UNION} \ (\text{«subquery»}) \\
  (\text{«subquery»}) \ \text{INTERSECT} \ (\text{«subquery»}) \\
  (\text{«subquery»}) \ \text{EXCEPT} \ (\text{«subquery»})
  \]

- The brackets are mandatory.
- The operands must be queries; you can’t simply use a relation name.
Example

(SELECT sid
 FROM Took
 WHERE grade > 95)
 UNION
(SELECT sid
 FROM Took
 WHERE grade < 50);
Operations $\cup$, $\cap$, and $-$ with Bags

- For $\cup$, $\cap$, and $-$ the number of occurrences of a tuple in the result requires some thought.
- (But it makes total sense.)

- (In-Class) Exercises:
  1. $\{1, 1, 1, 3, 7, 7, 8\} \cup \{1, 5, 7, 7, 8, 8\}$
  2. $\{1, 1, 1, 3, 7, 7, 8\} \cap \{1, 5, 7, 7, 8, 8\}$
  3. $\{1, 1, 1, 3, 7, 7, 8\} \setminus \{1, 5, 7, 7, 8, 8\}$
1. \( \{1, 1, 1, 3, 7, 7, 8\} \cup \{1, 5, 7, 7, 8, 8\} \)
   
   \[= \{1, 1, 1, 3, 7, 7, 8, 1, 5, 7, 7, 8, 8\}\]
   
   \[= \{1, 1, 1, 1, 3, 5, 7, 7, 7, 7, 8, 8, 8\}\]

1. \( \{1, 1, 1, 3, 7, 7, 8\} \cap \{1, 5, 7, 7, 8, 8\} \)
   
   \[= \{1, 7, 7, 8\}\]

1. \( \{1, 1, 1, 3, 7, 7, 8\} - \{1, 5, 7, 7, 8, 8\} \)
   
   \[= \{1, 1, 3\} \]
**Operations \( \cup, \cap, \text{ and } - \) with Bags**

- Suppose tuple \( t \) occurs
  - \( m \) times in relation \( R \), and
  - \( n \) times in relation \( S \).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of occurrences of ( t ) in result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( R \cap S )</td>
<td>( \min(m, n) )</td>
</tr>
<tr>
<td>( R \cup S )</td>
<td>( m + n )</td>
</tr>
<tr>
<td>( R - S )</td>
<td>( \max(m-n, 0) )</td>
</tr>
</tbody>
</table>
Bag vs Set Semantics: which is used

• We saw that a SELECT-FROM-WHERE statement uses bag semantics by default.
  • Duplicates are kept in the result.

• The set (INTERSECT / UNION / EXCEPT) operations use set semantics by default.
  • Duplicates are eliminated from the result.
Motivation: Efficiency

• When doing projection, it is easier not to eliminate duplicates.
  • Just work one tuple at a time.

• For intersection or difference, it is most efficient to sort the relations first.
  • At that point you may as well eliminate the duplicates anyway.

You can actually control both of them in SQL!
Controlling Duplicate Elimination

• We can force the result of a SFW query to be a set by using `SELECT DISTINCT ...`

• We can force the result of a set operation to be a bag by using `ALL`

```sql
(SELECT sid
 FROM Took
 WHERE grade > 95)
 UNION ALL
(SELECT sid
 FROM Took
 WHERE grade < 50);
```
---Q2 in Handout Part B---
(Set Operations)
Views
The idea

• A view is a relation defined in terms of stored tables (called base tables) and possibly also other views.
• Access a view like any base table.
• Two kinds of view:
  • Virtual: no tuples are stored; view is just a query for constructing the relation when needed.
  • Materialized: actually constructed and stored. Expensive to maintain!
• We’ll use only virtual views.
  • PostgreSQL did not support materialized views until version 9.3 (which we are not running).
Views - Example

Student

Logical Schema

Stu_ID  Stu_Name  Proj_ID

Physical Schema
Example: defining a virtual view

• A view for students who earned an 80 or higher in a CSC course.

CREATE VIEW topresults AS
SELECT firstname, surname, cnum
FROM Student, Took, Offering
WHERE
    Student.sid = Took.sid AND
    Took.oid = Offering.oid AND
    grade >= 80 AND dept = 'CSC';
Uses for views

• Break down a large query.
• Provide another way of looking at the same data, e.g., for one category of user.
Class Exercises - Cont

3. Find the sID of students who have earned a grade of 85 or more in some course, or who have passed a course taught by Atwood. Use views for the intermediate steps.

    create view High as
    (select sid from took where grade >= 85);

    create view HighAtwood as
    (select sid from Took, Offering
    where grade >= 50 and Took.oid = Offering.oid and instructor = 'Atwood');

    (select * from high)
    UNION
    (select * from highAtwood);
Class Exercises - Cont

4. Find all terms when csc369 was not offered.
   (No need to use Views!)

   (select term
    from Offering)
   except
   (select term
    from Offering
    where dept = 'csc' and cNum = 369);

ROSI Schema
Students(sID, surName, campus)
Courses(dept, cNum, cName, br)
Offerings(oID, dept, cNum, term, jnst)
Took(sID, oID, grade)