CompSci 516
Database Systems

Lecture 1
Introduction
and
SQL

Instructor: Sudeepa Roy
Course Website

• http://www.cs.duke.edu/courses/fall19/compsci516/

• Please check frequently for updates!
Instructor

• Sudeepa Roy
  – sudeepa@cs.duke.edu
  – https://users.cs.duke.edu/~sudeepa/
  – office hour: Tuesdays 2:45 pm – 3:45 pm, LSRC D325, and by appointments
  – No office hour today, instead on Friday 8/30 2-3 pm

• About myself
  – Assistant Professor in CS
  – PhD: UPenn, Postdoc: Univ. of Washington
  – Joined Duke CS in Fall 2015
  – Research interests:
    • Data Analysis, causality, query optimization, data science, database theory, applications of data, uncertain data,...
Three (half*--)TAs

• Yuchao Tao
  – yuchao.tao@duke.edu

• Yanlin Yu
  – yanlin.yu@duke.edu

• Tianrui Zhang
  – tanrui.zhang@duke.edu

• All CompSci 516 veterans!
  – office hours: TBD

* Offering you full help!
Logistics

• **Discussion forum: Piazza**
  – All enrolled students (by yesterday) are already there
  – Send me an email if you have not received a welcome email from Piazza

• **To reach course staff:**
  – compsci516-staff@cs.duke.edu
  – Please use piazza as much as possible

• **Lecture slides will be uploaded before the class as incomplete notes**
  – but will be updated after the class
Grading

- Three Homework: 30%
- Project: 10%
- Midterm: 20%
- Final: 30%
- Class participation: 10%
  - In-class quizzes: 5%
  - In-class labs: 5%
Grading Strategy

• Relative grading
  – The actual grade distribution at the end will depend on the performance of the entire class on all the components.
  – Topper of the class gets A+ irrespective of the number, and all and only “above expectation” performances get A+.
  – No fixed lowest grade or grade distribution.
  – Everyone can get good grade by working hard!
Homework

- Due in about 2 weeks after they are posted/previous hw is due
  - ALWAYS start early!
  - Part of the homework may be due in 1 week

- Two *late days* with penalty
  - For the take-home part (not the in-class lab part) of each homework
  - 25% penalty on the entire assignment if you submit within the next 24 hours after the deadline
  - 50% penalty on the entire assignment if you submit within the next 24 hours after the deadline
  - No credit after 48 hours
  - No credit after solutions are posted (even if within the first 48 hours)
  - Start early and do not count on late days!

- contact the instructor if you have a *valid* reason to be late
  - Another exam, project, hw is NOT a valid reason – we will always be fair to all

- To be done strictly individually

- PLEASE READ WHAT IS ALLOWED/NOT ALLOWED (will be repeated in class next week)

Homework Overview

• You will learn how to use traditional and new database systems in the homework
  – Have to learn them mostly on your own following tutorials available online and with some help from the TA

• HW1: Traditional DBMS
  – SQL and Postgres (and some XML too!)

• HW2: Distributed data processing
  – Spark and AWS

• HW3: NOSQL
  – MongoDB
Exams

• Midterm – Oct 15 (Tues)
• Final – Dec 14 (Sat)

• In class
• Closed book, closed notes, no electronic devices
• Total weight: 20 + 30 % = 50 %
• Exams will test your understanding of the material
• Both exams are comprehensive
  – would include every lecture up to the exams
Projects

• 10% weight
• In groups of 3-4
  – Groups of smaller and larger sizes need instructor’s permission
  – Each group member should do approx. equal work
• Very flexible in terms of topic!
• Show your creativity and researcher-side!
• Work done should be at least equivalent to
  – one hw * no. of group members

• All group members will get the same grade
• More information and ideas for projects will be posted later
Project Deliverables

1. Project proposal
   – problem selection is part of the project

2. Midterm progress report

3. Final project report

4. A final 5-10 mins project presentation and/or demonstration

• Due dates will be posted (about 1 month time for all three reports)
Class Participation

• 5% for quizzes, 5% for in-class labs
• Please bring laptops every day!
• Pop-up quiz
  – Participation (50%) + correct answering (50%)
  – lowest score will be dropped
• In-class labs
  – Attending the lab and submitting some solutions (50%)
  – Submitting correct solutions: within 24 hours after class ends (50%)
  – “Extra credit” 10% for submitting *all* correct solutions in class!
Please ask questions in class!

• **In general, actively participate in the class!**
  – Ask questions in class and on piazza
  – Stop me as many times as you need to understand the lectures
  – Answer each other’s questions on piazza

• Also send (anonymous or not) feedback, suggestions, or concerns on Piazza or by email
Reading Material

- Will mostly follow the "cowbook" by Ramakrishnan-Gehrke
  - The chapter numbers will be posted
- You do not have to buy the books, but it will be good to consult them from time to time
- You should be prepared to do quite a bit of reading from various books and papers
A Quick Survey

• Have you taken an undergrad database course earlier
  – CS 316/equivalent?

• Are you familiar with
  – SQL?
  – RA? ($\sigma$, $\Pi$, $\times$, $\bowtie$, $\rho$, $\cup$, $\cap$, $-$)
  – Keys, foreign keys?
  – Index in databases?
  – Logic: $\land$, $\lor$, $\forall$, $\exists$, $\neg$, $\in$, $\Rightarrow$
  – Transactions?
  – Map-reduce/Spark?
  – NOSQL?

• Have you ever worked with a dataset?
  – relational database, text, csv, XML

• Have you ever used a database system?
  – PostGres, MySQL, SQL Server, SQL Azure
What is this course about?

• This is a graduate-level database course in CS
  – We will cover principles, internals, and applications of database systems in depth

• Database concepts
  – Data Models, SQL, Views, Constraints, RA, Normalization

• Principles and internals of database management systems (DBMS)
  – Indexing, Query Execution-Algorithms-Optimization, Transactions, Parallel and Distributed Query Processing, Map Reduce

• Advanced and research topics in databases
  – e.g. Datalog, NOSQL, Data mining, ...
What this course is NOT about

- Spark, AWS, cluster computing...
  - Partially covered in a HW and a lecture
- Machine learning based analytics
- Statistical methods for data analytics
- Python, R, ...
Why should we care about databases?

• We are in a data-driven world

• Data = Currency, Data = Power, Data = Fun

• “Big Data” is supposed to change the mode of operation for almost every single field
  – Science, Technology, Healthcare, Business, Manufacturing, Journalism, Government, Education, ...

• We must know how to collect, store, process, and analyze such data

• Storing data in flat files and writing python or C code would fail at some point!

• And hundreds of jobs on data science, data analysis, data engineer, ...!
This week’s plan

• Today
  – Relational Data Model and SQL
• Lecture-2:
  – First In-class lab on SQL (conducted by Yanlin and Tianrui)
  – You will install postgres, work on MovieLens data on movie reviews, and then write some queries
  – Will be graded
    • You will submit solutions on Gradescope (auto-graded instantaneously!)
  – Do not forget your laptop!
    • Any platform should be fine
  – Feel free to attend even if you are on the waitlist and would like to enroll in this class

• Next week:
  – Data model and data independence, more SQL
Relational Data Model

• Proposed by Edward (Ted) Codd in 1970
  – won Turing award for it!

• Motivation:
  – Simplicity
  – Easy query optimizations
  – Separation of abstraction and operations
    • More next week
Relational Data Model

The data description construct is a Relation
- Represented as a “table”
- Basically a “set” of records (set semantic)
  - order does not matter
  - and all records are distinct

however, it is true for the relational model, not for standard DBM
- allow duplicate rows (bag semantic)
  - unless restricted by key constraints. Why?

<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>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith1@math</td>
<td>19</td>
<td>3.8</td>
</tr>
<tr>
<td>53831</td>
<td>Madayan</td>
<td>madayan@music</td>
<td>11</td>
<td>1.8</td>
</tr>
<tr>
<td>53832</td>
<td>Guldu</td>
<td>guldu@music</td>
<td>12</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Students

Bag: {1, 1, 2, 2, 3, 2, 1, 5, 6, 1}
Set: {1, 2, 3, 5, 6}
Why “bag semantic” and not “set semantic” in standard DBMSs?

- Primarily performance reasons
- Duplicate elimination is expensive (requires sorting)
- Some operations like “projection”’s are much more efficient on bags than sets
What is a poorly chosen attribute in this relation?

• Relational database = a set of relations
• A Relation : made up of two parts
  1. Schema
  2. Instance
Schema and Instance

- One schema can have multiple instances

**Schema:**
- A template for describing an entity/relationship (e.g. students)
- specifies name of relation + name and type of each column
  e.g. `Students(sid: string, name: string, login: string, age: integer, gpa: real)`.

**Instance:**
- When we fill in actual data values in a schema
- a table, has rows and columns
- each row/tuple follows the schema and domain constraints
- `#Rows = cardinality, #fields = degree or arity`
- example below

<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>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith1@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>

Cardinality = 3, degree = 5
SQL
(Structured Query Language)
Relational Query Languages

• A major strength of the relational model: supports simple, powerful querying of data.

• Queries can be written intuitively, and the DBMS is responsible for an efficient evaluation
  – The key: precise semantics for relational queries
  – Based on a sound theory!
  – Allows the optimizer to extensively re-order operations, and still ensure that the answer does not change.
The SQL Query Language

• Developed by IBM (systemR) in the 1970s based on Ted Codd’s relational model
  – First called “SEQUEL” (Structured English Query Language)
• First commercialized by Oracle (then Relational Software) in 1979
• Standards by ANSI and ISO since it is used by many vendors
  – SQL-86, -89 (minor revision), -92 (major revision), -96, -99 (major extensions), -03, -06, -08, -11, -16
Purposes of SQL

• **Data Manipulation Language (DML)**
  – Querying: SELECT-FROM-WHERE
  – Modifying: INSERT/DELETE/UPDATE *(next week)*

• **Data Definition Language (DDL)**
  – CREATE/ALTER/DROP *(next week)*
To find all 18 year old students, we can write:

```
SELECT * 
FROM Students S 
WHERE S.age=18
```

To find just names and logins, replace the first line:

```
SELECT S.name, S.login
```
What does the following query compute?

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade="A"
```
Querying Multiple Relations

• What does the following query compute?

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade="A"
```

Given the following instances of Enrolled and Students:

<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>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>

**Enrolled**

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

we get:

<table>
<thead>
<tr>
<th>S.name</th>
<th>E.cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Topology112</td>
</tr>
</tbody>
</table>
Basic SQL Query

```
SELECT [DISTINCT] <target-list>
FROM <relation-list>
WHERE <qualification>
```

- **relation-list** A list of relation names
  - possibly with a “range variable” after each name
- **target-list** A list of attributes of relations in relation-list
- **qualification** Comparisons
  - (Attr op const) or (Attr1 op Attr2)
  - where op is one of =, <, >, <=, >= combined using AND, OR and NOT
- **DISTINCT** is an optional keyword indicating that the answer should not contain duplicates
  - Default is that duplicates are not eliminated!
Conceptual Evaluation Strategy

- **Semantics** of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of `<relation-list>`
  - Discard resulting tuples if they fail `<qualifications>`
  - Delete attributes that are not in `<target-list>`
  - If `DISTINCT` is specified, eliminate duplicate rows

- **This strategy is probably the least efficient way to compute a query!**
  - An optimizer will find more efficient strategies to compute the same answers
Example of Conceptual Evaluation

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND R.bid=103
```

<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</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

What does this query return?
Example of Conceptual Evaluation

```
SELECT S.sname
FROM   Sailors S, Reserves R
WHERE  S.sid=R.sid AND R.bid=103
```

Step 1: Form “cross product” of Sailor and Reserves

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

Sailor

<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</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

Reserves

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

Duke CS, Fall 2016

CompSci 516: Data Intensive Computing Systems
Example of Conceptual Evaluation

```sql
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND R.bid=103
```

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

Step 2: Discard tuples that do not satisfy <qualification>
Example of Conceptual Evaluation

SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND R.bid=103

Step 3: Select the specified attribute(s)

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

Duke CS, Fall 2016
CompSci 516: Data Intensive Computing Systems
Recap

3. \[\text{SELECT } S\text{.sname}\]
1. \[\text{FROM } \text{Sailors} \ S, \text{Reserves} \ R\]
2. \[\text{WHERE } S\text{.sid}=R\text{.sid} \text{ AND } R\text{.bid}=103\]

Always start from “FROM” -- form cross product
Apply “WHERE” -- filter out some tuples (rows)
Apply “SELECT” -- filter out some attributes (columns)

Ques. Does this get evaluated this way in practice in a Database Management System (DBMS)?

No! This is conceptual evaluation for finding what is correct!
We will learn about join and other operator algorithms later
A Note on “Range Variables”

• Sometimes used as a short-name

• The previous query can also be written as:

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND bid=103
```

OR

```
SELECT sname
FROM Sailors, Reserves
WHERE Sailors.sid=Reserves.sid
AND bid=103
```

It is good style, however, to use range variables always!
A Note on “Range Variables”

• Really needed only if the same relation appears twice in the FROM clause (called self-joins)

• Find pairs of Sailors of same age

```
SELECT  S1.sname, S2. name
FROM     Sailors S1, Sailors S2
WHERE    S1.age = S2.age AND S1.sid < S2.sid
```

Why do we need the 2nd condition?
Find sailor ids who’ve reserved at least one boat

```
SELECT ??
FROM Sailors S, Reserves R
WHERE S.sid = R.sid
```
Find sailor ids who’ve reserved at least one boat

Would adding `DISTINCT` to this query make a difference?

```
SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid = R.sid
```
Find sailors who’ve reserved at least one boat

```sql
SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid = R.sid
```

- Would adding `DISTINCT` to this query make a difference?
  - Note that if there are multiple bids for the same sid, you get multiple output tuples for the same sid
  - Without distinct, you get them multiple times

- What is the effect of replacing `S.sid` by `S.sname` in the `SELECT` clause?
  - Would adding `DISTINCT` to this variant of the query make a difference even if one sid reserves at most one bid?

**Sailor**

<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</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

**Reserves**

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Simple Aggregate Operators

Check yourself:
What do these queries compute?

```
SELECT COUNT (*)
FROM Sailors S
```

```
SELECT AVG (S.age)
FROM Sailors S
WHERE S.rating=10
```

```
SELECT COUNT (DISTINCT S.rating)
FROM Sailors S
WHERE S.sname='Bob'
```

```
SELECT COUNT ( [DISTINCT] A)
```

```
SUM ( [DISTINCT] A)
```

```
AVG ( [DISTINCT] A)
```

```
MAX (A)
```

```
MIN (A)
```

```
SELECT S.sname
FROM Sailors S
WHERE S.rating= (SELECT MAX(S2.rating)
                     FROM Sailors S2)
```

```
SELECT AVG ( DISTINCT S.age)
FROM Sailors S
WHERE S.rating=10
```

Check yourself:
What do these queries compute?
Next: different types of joins

- Theta-join
- Equi-join
- Natural join
- Outer Join
### Condition/Theta Join

```
SELECT *  
FROM  Sailors S, Reserves R  
WHERE  S.sid=R.sid and age >= 40
```

Form cross product, discard rows that do not satisfy the condition

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

**Equi Join**

A special case of theta join
Join condition only has equality predicate =

```
SELECT * 
FROM Sailors S, Reserves R 
WHERE S.sid = R.sid and age = 45
```
**Natural Join**

A special case of equi join
Equality condition on ALL common predicates (sid)
Duplicate columns are eliminated

```
SELECT * FROM Sailors S NATURAL JOIN Reserves R
```

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Outer Join

```
SELECT S.sid, R. bid
FROM Sailors S LEFT OUTER JOIN Reserves R
ON S.sid=R.sid
```

Preserves all tuples from the left table whether or not there is a match
if no match, fill attributes from right with null
Similarly RIGHT/FULL outer join
Expressions and Strings

SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM Sailors S
WHERE S.sname LIKE 'B_%B'

• Illustrates use of arithmetic expressions and string pattern matching
• Find triples (of ages of sailors and two fields defined by expressions) for sailors
  – whose names begin and end with B and contain at least three characters
• LIKE is used for string matching. `_` stands for any one character and `%` stands for 0 or more arbitrary characters
  – You will need these often
Find sid’s of sailors who’ve reserved a red or a green boat

- **UNION**: Can be used to compute the union of any two *union-compatible* sets of tuples
  - can themselves be the result of SQL queries
- If we replace **OR** by **AND** in the first version, what do we get?
- Also available: **EXCEPT** (What do we get if we replace **UNION** by **EXCEPT**?)

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND (B.color='red' OR B.color='green')
```

```sql
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND B.color='green'
```
Find sid’s of sailors who’ve reserved a red and a green boat
Find sid’s of sailors who’ve reserved a red and a green boat

- **INTERSECT:** Can be used to compute the intersection of any two union-compatible sets of tuples.

  - Included in the SQL/92 standard, but some systems don’t support it

```
SELECT S.sid
FROM Sailors S, Boats B1, Reserves R1,
    Boats B2, Reserves R2
WHERE S.sid=R1.sid AND R1.bid=B1.bid
    AND S.sid=R2.sid AND R2.bid=B2.bid
    AND (B1.color='red' AND B2.color='green')
```

```
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
    AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
WHERE S.sid=R.s.id AND R.bid=B.bid
    AND B.color='green'
```
Nested Queries

Find names of sailors who’ve reserved boat #103:

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                 FROM Reserves R
                 WHERE R.bid=103)
```

- **A very powerful feature of SQL:**
  - a WHERE/FROM/HAVING clause can itself contain an SQL query
- **To find sailors who’ve not reserved #103, use NOT IN.**
- **To understand semantics of nested queries, think of a nested loops evaluation**
  - For each Sailors tuple, check the qualification by computing the subquery
Nested Queries with Correlation

Find names of sailors who’ve reserved boat #103:

```
SELECT  S.sname  
FROM    Sailors S  
WHERE   EXISTS (SELECT *  
                FROM   Reserves R  
                WHERE  R.bid=103 AND S.sid=R.sid)
```

- **EXISTS** is another set comparison operator, like **IN**
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple
Nested Queries with Correlation

Find names of sailors who’ve reserved boat #103 at most once:

```
SELECT  S.sname
FROM    Sailors S
WHERE   UNIQUE (SELECT  R.bid
                 FROM    Reserves R
                 WHERE   R.bid=103 AND S.sid=R.sid)
```

- **If UNIQUE** is used, and * is replaced by *R.bid*, finds sailors with at most one reservation for boat #103
  - **UNIQUE** checks for duplicate tuples
More on Set-Comparison Operators

• We’ve already seen **IN, EXISTS and UNIQUE**

• Can also use **NOT IN, NOT EXISTS and NOT UNIQUE**.

• Also available: **op ANY, op ALL, op IN**
  – where op : >, <, =, <=, >=

• Find sailors whose rating is greater than that of some sailor called Horatio
  
  – similarly **ALL**

```
SELECT *
FROM Sailors S
WHERE S.rating > ANY (SELECT S2.rating
FROM Sailors S2
WHERE S2.sname=‘Horatio’)  
```
Summary

• Relational Data
• SQL
  – Semantic
  – Join
  – Simple Aggregates
  – Nested Queries

• You will learn these further and run yourself on PostGres on Thursday in the in-class lab on SQL!