CompSci 516
Database Systems

Lecture 1
Introduction and SQL

Instructor: Sudeepa Roy

Duke CS, Spring 2022
CompSci 516: Database Systems

Course Website

• http://www.cs.duke.edu/courses/spring22/compsci516/

• Please check frequently for updates!

Instructor

• Sudeepa Roy
  – sudeepa@cs.duke.edu
  – https://users.cs.duke.edu/~sudeepa/

• 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 TAs

• Yuxi Liu (grad TA)
• Shweta Patwa (grad TA)
• Joon Young Lee (UTA)

Duke CS, Spring 2022
CompSci 516: Database Systems

What are the goals of this course?

• Learn about “Database Systems” or Data Management in general

Why do we care about data? (easy)

Data = Money
Information
Power
Fun
in
Science, Business, Politics, Security
Sports, Education, ….
Wait... don’t we need to take a Machine Learning or Statistics course for those things? Yes, but...

... we also need to manage this (huge or not-so-huge) data!

Also think about building any application based on data from scratch
- E.g., your own version of mini-Amazon or a Book Selling Platform
- Large data! (think about all books in the world or even in English)
  - How do we start?
  - A short background survey first...

Who are the key people? (book-selling website)

Who are the key people? (book-selling website)

What should the user be able to do?
- i.e. what the interface look like? (think about Amazon)
What should the user be able to do?

- i.e. what the interface look like? (think about Amazon)

What should the platform do?

What are the desired and necessary properties of the platform?

That was the design phase (a basic one though)

How about C++, Java, or Python?

On data stored in large files
Sounds simple!

- Text files – for books, customer, ...
- Books listed with title, author, price, and no. of copies
- Fields separated by #’s

Query by programming

- James Morgan wants to buy "To Kill a Mockingbird"
- A simple script
  - Scan through the books file
  - Look for the line containing "To Kill a Mockingbird"
  - Check if the no. of copies is >= 1
  - Bill James $7.20 and reduce the no. of copies by 1

Better idea than scanning?

What if he changes the "query" and wants to buy a book by Victor Hugo?

Revisit: What are the desired and necessary properties of the platform?

- Should be able to handle a large amount of data
- Should be efficient and easy to use (e.g., search with authors as well as title)
- If there is a crash or loss of power, information should not be lost or inconsistent
  - Imagine a user was in the middle of a transaction when a crash happened, paid the money, but the book has not been purchased
- No surprises with multiple users logged in at the same time
  - Imagine one last copy of a book that two users are trying to purchase at the same time
- Easy to update and program
  - For the admin

Solution?

- DBMS = Database Management System

A DBMS helps the big ones!

Optimization
- We will learn these in the course!

Index
- Recovery
- Concurrency Control
- Declarative

DBMS helps the big ones!

Note: Not always the “standard” DBMS (called Relational DBMS), but we need to know pros and cons of all alternatives
CompSci 516: how database systems work and can be used by users

• This is a graduate-level database course in CS
  – We will cover principles, internals, and applications of database systems
• How can a user use a DBMS (programmer’s/designer’s perspective)
  – Run queries, update data, views (SQL, Relational Algebra, Relational calculus)
  – Design a good database (normalization, constraints)
  – Use different types of data (Mostly relational, also XML/JSON)
• How does a DBMS work (system’s or admin’s perspective, also for programmers for writing better queries)
  – Storage, index, query processing, join algorithms, query optimizations
  – Transactions: recovery and concurrency control
• Glimpse of advanced topics and other DBMS
  – NOSQL, Spark (big data), data mining, Datalog/recursive queries, Parallel and distributed DBMS

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
• Data mining...

Warnings!

• Course-load: Three solid programming-heavy and problem-solving HW on three systems + project + quizzes/labs + midterm + final
• HWs will be mostly “disjoint” from lectures and largely self-taught
  – use online tutorials and help of TAs, save enough time
  – HW1: Traditional DBMS – SQL and Postgres (and some XML too)
  – HW2: Distributed data processing – Spark and AWS
  – HW3: NOSQL, MongoDB
• Lecture materials are tested in exams
  – Make sure that you follow the lectures before the next class
  – Some practice problems in quizzes and in class
• We will assume backgrounds in some programming language, and in algorithms, data structures, sets, basic systems
  – If you are from a non-CS background, would need to learn these as needed quickly
• There will be some “theory”, algorithms, maths, and proofs, e.g.,
  – Relational calculus ≡ First-order logic (V, /, ∧, ∨, =)
  – Database normal forms, repairs (recursive queries), directed cyclic graphs (transactions)

Logistics

• Discussion forum: Ed
  – All enrolled students are already there
• To reach course staff:
  – Private message on Ed to reach all staff, or send instructor email only for logistics
• Lecture slides will be uploaded before the class as incomplete notes
  – Will be updated after the class
• Other tools
  – Gradescope (already enrolled)
  – Sakai
  – Gradiance (code will be sent)

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 might be good to consult them from time to time

Grading

• Three Homework: 30%
• Group Project: 13%
• Midterm: 20%
• Final: 25%
  – Exams are comprehensive, closed book/notes, no collaboration
• Class participation: 12%
  – Quizzes/labs in class or short deadline (lowest score dropped): 10%
  – Communication: 2%

Please bring laptops every day!
Grading Strategy

- Absolute and adjustable grading
  - 90% or above: Guaranteed A-range grades (A-, A, or A+)
  - 80-90%: Guaranteed B-range grades (B-, B, or B+)
  - Etc. (see website)
- Thresholds may slide down (your grade may go up) based on class performance
- +/- will depend on the class performance
  - Topper of the class gets A+, and all and only “above expectation” performances get A+
- 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
- Late policy:
  - 5% penalty per hour unless there are valid reasons or permissions
  - Check out website
  - 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 individually, but discussions allowed
  - More details in the next class

Projects

- 13% weight
- In groups of 4
  - Groups of smaller and larger sizes need instructor’s permission
- Flexible in terms of topic related to data management
  - Fixed option: Database-backed website (template provided)
  - Open options (needs to be approved by the instructor): analyzing data, any db-related research problem
  - Running a standard ML classification/regression on a dataset is not enough
- Work done should be at least equivalent to
  - 1.3 hw * no. of group members
- Weekly updates + proposal + midterm report + final report + demo/presentation
- More information and ideas for projects will be posted later

Some Important Dates

- January 20:
  - Team members’ names + tentative project topic due
- February 1:
  - Project proposal due
- February 15:
  - Midterm
  - HW1 will be released next week

Please ask questions in class!

- In general, actively participate in the class!
  - Ask questions in class and on Ed
  - Stop me as many times as you need to understand the lectures
  - Answer each other’s questions on Ed – this is a big class and getting answers from TAs may take time, although we will monitor all threads
- Anonymous feedback form link on Ed
  - To be checked at least once weekly
  - All feedback, suggestions, concerns welcome!

Let’s get started!

Relational Data Model

What is a good model to store data? Tree? Nested data? Graph?

(just) Tables!
Edgar F. Codd (1923-2003)
- Pilot in the Royal Air Force in WW2
- Inventor of the relational model and algebra while at IBM (1970)
- Turing Award, 1981

RDBMS = Relational DBMS

Motivation of relational model
- Simplicity
- Easy query optimizations
- Separation of abstraction and operations

Relational Data Model

<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>

- 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?

Bag vs. Set

<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>

- Why “bag semantic” and not “set semantic” in standard DBMSs?

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)

See the separate instructions to install postgres to practice!
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)

Querying Multiple Relations

- What does the following query compute?

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>

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

we get: ??

The SQL Query Language

- To find all 18 year old students, we can write:

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

<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>

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

```sql
SELECT S.name, S.login
```

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>

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

we get: ??

The SQL Query Language

- To find all 18 year old students, we can write:

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

<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>

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

```sql
SELECT S.name, S.login
```
Basic SQL Query

SELECT [DISTINCT] <target-list> FROM <relations-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: an optional keyword indicating that the answer should not contain duplicates
  - Default is that duplicates are not eliminated!

Example of Conceptual Evaluation

Sailor

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

Sailor

SELECT s.sname FROM sailors s, reserves r WHERE s.sid=r.sid AND r.bid=103

Step 2: Discard tuples that do not satisfy <qualifications>

Sailor

SELECT s.sname FROM sailors s, reserves r WHERE s.sid=r.sid AND r.bid=103

Step 3: Select the specified attribute(s)

Conceptual Evaluation Strategy

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

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of <relations-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
Recap

3. SELECT S.sname
1. FROM Sailors S, Reserves R
2. WHERE S.sid=R.sid AND R.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 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

<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>

Find sailor ids who’ve reserved at least one boat

| SELECT ??? FROM Sailors S, Reserves R WHERE S.sid=R.sid

<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>

Find sailors who’ve reserved at least one boat

| SELECT ??? FROM Sailors S, Reserves R WHERE S.sid=R.sid

<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>

• Would adding DISTINCT to this query make a difference?

• What is the effect of replacing S.sid by S.sname in the SELECT clause?
Simple Aggregate Operators

- **SELECT COUNT(*)**
  FROM Sailors S
- **SELECT AVG(Sage)**
  FROM Sailors S
- **SELECT COUNT(DISTINCT S.age)**
  FROM Sailors S
- **SELECT COUNT(*)**
  FROM Sailors S
- **SELECT S.name**
  FROM Sailors S
  WHERE Szrating = 10
- **SELECT AVG(DISTINCT S.age)**
  FROM Sailors S
  WHERE Szrating = 10

Next: different types of joins

- **Theta-join**
- **Equi-join**
- **Natural join**
- **Outer Join**

Condition/Theta Join

- Form cross product, discard rows that do not satisfy the condition
- Equality condition on ALL common predicates (sid)
- Duplicate columns are eliminated

<table>
<thead>
<tr>
<th>sid</th>
<th>name</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>22</td>
<td>dustin</td>
<td>7</td>
<td>55</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>35</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

Equi Join

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

<table>
<thead>
<tr>
<th>sid</th>
<th>name</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>22</td>
<td>dustin</td>
<td>7</td>
<td>55</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>35</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

Natural Join

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

<table>
<thead>
<tr>
<th>sid</th>
<th>name</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>22</td>
<td>dustin</td>
<td>7</td>
<td>55</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>35</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

Outer Join

- 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

<table>
<thead>
<tr>
<th>sid</th>
<th>name</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>22</td>
<td>dustin</td>
<td>7</td>
<td>55</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>35</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>
Expressions and Strings

- 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

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

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
- 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
Sailors (sid, sname, rating, age)
Reserves (sid, bid, day)
Boats (bid, bname, color)
```

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

- `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

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

Nested Queries with Correlation

- `EXISTS` is another set comparison operator, like `IN`
- Illustrates why, in general, subquery must be recomputed for each Sailors tuple

```sql
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
  FROM Reserves R
  WHERE R.bid = 103 AND S.sid = R.sid)
```
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: \( \text{op ANY, op ALL, op IN} \)
  - where \( \text{op} \) : \( >, <, \leq, \geq \)
- Find sailors whose rating is greater than that of some sailor called Horatio
  - similarly ALL

Summary

- Relational Data
- SQL
  - Semantic
  - Join
  - Simple Aggregates
  - Nested Queries