Life is just a bowl of queries.
~Anon

Administrivia
- Midterm 1 was a bit easier than I wanted it to be.
  - Mean was 80
  - Three people got 100(!)
  - I’m actually quite pleased.
  - But, I do plan to “kick it up a notch” for the future exams.
- Be sure to register your name with your cs186 login if you haven’t already — else, you risk not getting grades.
- Homework 2 is being released today.
  - Today and Tuesday’s lectures provide background.
  - HW 2 is due Tuesday 3/14
  - It’s more involved than HW 1.

Relational Query Languages
- A major strength of the relational model: supports simple, powerful querying of data.
- Two sublanguages:
  - DDL - Data Definition Language
    - define and modify schema (at all 3 levels)
  - DML - Data Manipulation Language
    - Queries can be written intuitively.
- The DBMS is responsible for efficient evaluation.
  - The key: precise semantics for relational queries.
  - Allows the optimizer to extensively re-order operations, and still ensure that the answer does not change.
  - Internal cost model drives use of indexes and choice of access paths and physical operators.

The SQL Query Language
- The most widely used relational query language.
- Originally IBM, then ANSI in 1986
- Current standard is SQL-2003
  - Introduced XML features, window functions, sequences, auto-generated IDs.
  - Not fully supported yet
- SQL-1999 Introduced “Object-Relational” concepts. Also not fully supported yet.
- SQL92 is a basic subset
  - Most systems support a medium
  - PostgreSQL has some “unique” aspects (as do most systems).

DDL – Create Table
- CREATE TABLE table_name
  ( { column_name data_type [ DEFAULT default_expr ] [ column_constraint [ , ... ] ] |
    table_constraint [ , ... ] } )

- Data Types (PostgreSQL) include:
  - character(n) – fixed-length character string
  - character varying(n) – variable-length character string
  - smallint, integer, bigint, numeric, real, double precision
  - date, time, timestamp, …
  - serial - unique ID for indexing and cross reference
  - serial - unique ID for indexing and cross reference
- PostgreSQL also allows OIDs and other “system types”, arrays, inheritance, rules...
  - conformance to the SQL-1999 standard is variable.

Constraints
- Recall that the schema defines the legal instances of the relations.
- Data types are a way to limit the kind of data that can be stored in a table, but they are often insufficient.
  - e.g., prices must be positive values
  - uniqueness, referential integrity, etc.
- Can specify constraints on individual columns or on tables.
Column constraints

[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | UNIQUE | PRIMARY KEY | CHECK (expression) | REFERENCES ref_table [ ( ref_column ) ] [ ON DELETE action ] [ ON UPDATE action ]}

primary key = unique + not null; also used as default target for references. (can have at most 1)
expression must produce a boolean result and reference that column’s value only.
references is for foreign keys; action is one of: NO ACTION, CASCADE, SET NULL, SET DEFAULT

Table constraints

• CREATE TABLE table_name
  ( { column_name data_type [ DEFAULT default_expr ] | column_constraint [ , ... ] | table_constraint [ , ... ] })

Table Constraints:

• CONSTRAINT constraint_name
  ( UNIQUE ( column_name [ , ... ] ) | PRIMARY KEY ( column_name [ , ... ] ) | CHECK ( expression ) | FOREIGN KEY ( column_name [ , ... ] ) REFERENCES ref_table [ ( ref_column ) ] [ ON DELETE action ] [ ON UPDATE action ]}

Here, expressions, etc can include multiple columns

Create Table (Examples)

CREATE TABLE films (  
code CHAR(5) PRIMARY KEY,  
title VARCHAR(40),  
did DECIMAL(3),  
date_prod DATE,  
kind VARCHAR(10),  
CONSTRAINT production UNIQUE(date_prod)  
FOREIGN KEY did REFERENCES distributors  
ON DELETE NO ACTION  
);
CREATE TABLE distributors (  
did DECIMAL(3) PRIMARY KEY,  
name VARCHAR(40)  
CONSTRAINT con1 CHECK (did > 100 AND name <> ' ')  
);

Other DDL Statements

• Alter Table
  - use to add/remove columns, constraints, rename things ...

• Drop Table
  - Compare to “Delete * From Table”

• Create/ Drop View
• Create/ Drop Index

• Grant/ Revoke privileges
  - SQL has an authorization model for saying who can read/modify/delete etc. data and who can grant and revoke privileges!

The SQL DML

• Single-table queries are straightforward.

• 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

Querying Multiple Relations

• Can specify a join over two tables as follows:

SELECT S.name, E.cid  
FROM Students S, Enrolled E  
WHERE S.sid=E.sid AND E.grade='B'

Note: obviously no referential integrity constraints have been used here.
Basic SQL Query

- **relation-list**: A list of relation names
  - possibly with a `range-variable` after each name
- **target-list**: A list of attributes of tables in `relation-list`
- **qualification**: Comparisons combined using AND, OR and NOT.
  - Comparisons are `Attr op const or Attr1 op Attr2`, where `op` is one of `= <> <= > >=`
- **DISTINCT**: optional keyword indicating that the answer should not contain duplicates.
  - In SQL SELECT, the default is that duplicates are not eliminated! (Result is called a "multiset")

### Query Semantics

- Semantics of an SQL query are defined in terms of the following conceptual evaluation strategy:
  1. do FROM clause: compute cross-product of tables (e.g., Students and Enrolled).
  2. do WHERE clause: Check conditions, discard tuples that fail. (i.e., "selection").
  3. do SELECT clause: Delete unwanted fields. (i.e., "projection").
  4. If DISTINCT specified, eliminate duplicate rows.

Probably the least efficient way to compute a query!
- An optimizer will find more efficient strategies to get the same answer.

### Cross Product

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

**Step 2)** Discard tuples that fail predicate

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

**Step 3)** Discard Unwanted Columns

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

**Now the Details**

We will use these instances of relations in our examples.

- **Reserves**
  - `sid` `bid` `day`
  - 22 101 10/10/96
  - 95 103 11/12/96

- **Sailors**
  - `sid` `name` `rating` `age`
  - 22 Dustin 7 45.0
  - 31 Lubber 8 55.5
  - 95 Bob 3 63.5

- **Boats**
  - `bid` `bname` `color`
  - 101 Interlake blue red
  - 102 Interlake red
  - 103 Clipper green red
**Example Schemas (in SQL DDL)**

```sql
CREATE TABLE Sailors (sid INTEGER, sname CHAR(20), rating INTEGER, age REAL, PRIMARY KEY sid)

CREATE TABLE Boats (bid INTEGER, bname CHAR(20), color CHAR(10), PRIMARY KEY bid)

CREATE TABLE Reserves (sid INTEGER, bid INTEGER, day DATE, PRIMARY KEY (sid, bid, day), FOREIGN KEY sid REFERENCES Sailors, FOREIGN KEY bid REFERENCES Boats)
```

**Another Join Query**

```sql
SELECT sname
FROM     Sailors, Reserves
WHERE  Sailors.sid=Reserves.sid
AND 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>45.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
<td>95</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

**Some Notes on Range Variables**

- Can associate “range variables” with the tables in the FROM clause.
  - saves writing, makes queries easier to understand
- Needed when ambiguity could arise.
  - for example, if same table used multiple times in same FROM (called a “self-join”)

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

Can be rewritten using range variables as:

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

**More Notes**

- Here’s an example where range variables are required (self-join example):

```sql
SELECT x.sname, x.age, y.sname, y.age
FROM Sailors x, Sailors y
WHERE  x.age > y.age
```

- Note that target list can be replaced by “*” if you don’t want to do a projection:

```sql
SELECT * FROM Sailors x
WHERE x.age > 20
```

**Expressions**

- Can use arithmetic expressions in SELECT clause (plus other operations we’ll discuss later)
- Use AS to provide column names

```sql
SELECT S.age, S.age-5 AS age1, 2*S.age AS age2
FROM  Sailors S
WHERE  S.sname = 'dustin'
```

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

```sql
SELECT S1.sname AS name1, S2.sname AS name2
FROM  Sailors S1, Sailors S2
WHERE  2*S1.rating = S2.rating - 1
```
String operations

- SQL also supports some string operations
- "LIKE" is used for string matching.

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

`_` stands for any one character and `%` stands for 0 or more arbitrary characters.

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 (which are themselves the result of SQL queries).

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

Vs.

```sql
```

AND Continued...

- INTERSECT: discussed in book. Can be used to compute the intersection of any two union-compatible sets of tuples.
- Also in text: EXCEPT (sometimes called MINUS)
- Included in the SQL/92 standard, but many systems don’t support them.

```sql
```

Nested Queries

- Powerful feature of SQL: WHERE clause can itself contain an SQL query!
  - Actually, so can FROM and HAVING clauses.

```sql
SELECT R1.sid
FROM Boats B1, Reserves R1, Boats B2, Reserves R2
```

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

- EXISTS is another set comparison operator, like IN.
- Can also specify NOT EXISTS
- 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 in a subquery:
  - Subquery must be recomputed for each Sailors tuple.
  - Think of subquery as a function call that runs a query!
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: \textit{op ANY}, \textit{op ALL}
- Find sailors whose rating is greater than that of some sailor called Horatio:

\[
\text{SELECT } * \\
\text{FROM Sailors S} \\
\text{WHERE S.rating > ANY (SELECT S2.rating} \\
\text{FROM Sailors S2} \\
\text{WHERE S2.sname='Horatio')} 
\]

Rewriting INTERSECT Queries Using IN

Find sid's of sailors who've reserved both a red and a green boat:

\[
\text{SELECT R.sid} \\
\text{FROM Boats B, Reserves R} \\
\text{WHERE R.bid=B.bid} \\
\text{AND B.color='red'} \\
\text{AND R.sid IN (SELECT R2.sid} \\
\text{FROM Boats B2, Reserves R2} \\
\text{WHERE R2.bid=B2.bid} \\
\text{AND B2.color='green')} 
\]

- Similarly, EXCEPT queries re-written using NOT IN.
- How would you change this to find \textit{names (not sid's)} of Sailors who've reserved both red and green boats?

Division in SQL

Find names of sailors who've reserved all boats.

- Example in book, not using EXCEPT:

\[
\text{SELECT S.sname} \\
\text{Sailors S} \\
\text{WHERE NOT EXISTS (SELECT B.bid} \\
\text{FROM Boats B} \\
\text{WHERE NOT EXISTS (SELECT R.bid} \\
\text{FROM Reserves R} \\
\text{WHERE R.bid=B.bid} \\
\text{AND R.sid=S.sid))} 
\]

Basic SQL Queries - Summary

- An advantage of the relational model is its well-defined query semantics.
- SQL provides functionality close to that of the basic relational model.
  - some differences in duplicate handling, null values, set operators, etc.
- Typically, many ways to write a query
  - the system is responsible for figuring a fast way to actually execute a query regardless of how it is written.
  - Lots more functionality beyond these basic features.

Aggregate Operators

- Significant extension of relational algebra.

\[
\text{SELECT COUNT (S.rating)} \\
\text{FROM Sailors S} \\
\text{WHERE S.rating=10} \\
\text{SELECT COUNT (DISTINCT S.rating)} \\
\text{FROM Sailors S} \\
\text{WHERE S.sname='Bob'} 
\]
Aggregate Operators (continued)

- COUNT (*)
- COUNT ([DISTINCT] A)
- SUM ([DISTINCT] A)
- AVG ([DISTINCT] A)
- MAX (A)
- MIN (A)

single column

Find name and age of the oldest sailor(s)

- The first query is incorrect!

- Third query equivalent to second query
  - allowed in SQL/92 standard, but not supported in some systems.

GROUP BY and HAVING

- So far, we've applied aggregate operators to all (qualifying) tuples.
  - Sometimes, we want to apply them to each of several groups of tuples.
- Consider: 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 \( i = 1, 2, \ldots, 10 \):

- \( \text{SELECT MIN (S.age)} \)
- \( \text{FROM Sailors S} \)
- \( \text{WHERE S.rating} = i \)

Queries With GROUP BY

- To generate values for a column based on groups of rows, use aggregate functions in SELECT statements with the GROUP BY clause

GROUP BY Examples

For each rating, find the average age of the sailors

- \( \text{SELECT S.rating, AVG (S.age)} \)
- \( \text{FROM Sailors S} \)
- \( \text{GROUP BY S.rating} \)

For each rating find the age of the youngest sailor with age \( \geq 18 \)

- \( \text{SELECT S.rating, MIN (S.age)} \)
- \( \text{FROM Sailors S} \)
- \( \text{WHERE S.age} \geq 18 \)
- \( \text{GROUP BY S.rating} \)

Conceptual Evaluation

- The cross-product of \( \text{relation-list} \) is computed, tuples that fail \( \text{qualification} \) are discarded, “unnecessary” fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in \( \text{grouping-list} \).

- One answer tuple is generated per qualifying group.
### Queries With GROUP BY and HAVING

**Find the number of reservations for each red boat.**

```sql
SELECT B.bid, COUNT(*) AS scount
FROM Boats B, Reserves R
WHERE R.bid = B.bid
AND B.color = 'red'
GROUP BY B.bid
```

- Grouping over a join of two relations.

**Find the age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors.**

```sql
SELECT S.rating, MIN(S.age)
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1
```

- Form groups as before.
- The `group-qualification` is then applied to eliminate some groups.
  - Expressions in `group-qualification` must have a **single value per group**.
  - That is, attributes in `group-qualification` must be arguments of an aggregate op or must also appear in the `grouping-list`. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

**Conceptual Evaluation**

- Form groups as before.
- The `group-qualification` is then applied to eliminate some groups.
  - Expressions in `group-qualification` must have a **single value per group**.
- That is, attributes in `group-qualification` must be arguments of an aggregate op or must also appear in the `grouping-list`. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

---

**Answer relation**

<table>
<thead>
<tr>
<th>rating</th>
<th>m-age</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>45.0</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Answer Table**

<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>71</td>
<td>zorba</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

1. Form cross product
2. Delete unneeded columns, rows, form groups
Find names of sailors who've reserved all boats.

- Example in book, not using EXCEPT:

```sql
SELECT S.name
FROM Sailors S
WHERE NOT EXISTS (SELECT B.bid
FROM Boats B
WHERE NOT EXISTS (SELECT R.bid
FROM Reserves R
WHERE R.bid=B.bid
AND R.sid=S.sid))
```

Sailors S such that there is no boat B without a Reserves tuple showing S reserved B

Find names of sailors who've reserved all boats.

- Can you do this using Group By and Having?

```sql
SELECT S.name
FROM Sailors S, reserves R
WHERE S.sid = R.sid
GROUP BY S.name, S.sid
HAVING COUNT(DISTINCT R.bid) = (Select COUNT (*) FROM Boats)
```

Note: must have both sid and name in the GROUP BY clause. Why?

---

INSERT

```sql
INSERT [INTO] table_name [(column_list)]
VALUES ( value_list)
```

```sql
INSERT INTO Boats VALUES (105, 'Clipper', 'purple')
INSERT INTO Boats (bid, color) VALUES (99, 'yellow')
```

You can also do a “bulk insert” of values from one table into another:

```sql
INSERT INTO TEMP(bid) SELECT r.bid FROM Reserves R WHERE r.sid = 22;
```

(must be type compatible)

---

DELETE & UPDATE

```sql
DELETE [FROM] table_name
[WHERE  qualification]
```

DELETE FROM Boats WHERE color = 'red'

DELETE FROM Boats b
WHERE b. bid = (SELECT r.bid FROM Reserves R WHERE r.sid = 22)

Can also modify tuples using UPDATE statement.

```sql
UPDATE Boats
SET Color = "green"
WHERE bid = 103;
```

---

Null Values

- Field values in a tuple are sometimes unknown (e.g., a rating has not been assigned) or inapplicable (e.g., no spouse’s name).
  - SQL provides a special value `null` for such situations.
- The presence of null complicates many issues. E.g.:
  - Special operators needed to check if value is/is not null
  - Is rating>8 true or false when rating is equal to null? What about AND, OR and NOT connectives?
  - We need a 3-valued logic (true, false and unknown).
  - Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don’t evaluate to true.)
  - New operators (in particular, outer joins) possible/needed.
Joins

SELECT (column_list) 
FROM table_name 
[INNER | {LEFT | RIGHT | FULL } OUTER] JOIN table_name 
ON qualification_list 
WHERE ...

Explicit join semantics needed unless it is an INNER join (INNER is default)

Inner Join

Only the rows that match the search conditions are returned.

```
SELECT s.sid, s.name, r.bid 
FROM Sailors s INNER JOIN Reserves r 
ON s.sid = r.sid
```

Returns only those sailors who have reserved boats

SQL-92 also allows:

```
SELECT s.sid, s.name, r.bid 
FROM Sailors s NATURAL JOIN Reserves r 
```

“NATURAL” means equi-join for each pair of attributes with the same name (may need to rename with “AS”)

```
SELECT s.sid, s.name, r.bid 
FROM Sailors s INNER JOIN Reserves r 
ON s.sid = r.sid
```

```
<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>55.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>63.5</td>
</tr>
</tbody>
</table>
```

```
s.sid  s.name  r.bid
-------  ------  ----
 22      Dustin  101
 95      Bob    103
```

Left Outer Join

Left Outer Join returns all matched rows, plus all unmatched rows from the table on the left of the join clause

( use nulls in fields of non-matching tuples)

```
SELECT s.sid, s.name, r.bid 
FROM Sailors s LEFT OUTER JOIN Reserves r 
ON s.sid = r.sid
```

Returns all sailors & information on whether they have reserved boats

```
<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>55.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>63.5</td>
</tr>
</tbody>
</table>
```

```
s.sid  s.name  r.bid
-------  ------  ----
 22      Dustin  101
 95      Bob    103
 31      Lubber
```

Right Outer Join

Right Outer Join returns all matched rows, plus all unmatched rows from the table on the right of the join clause

```
SELECT r.sid, b.bid, b.name 
FROM Reserves r RIGHT OUTER JOIN Boats b 
ON r.bid = b.bid
```

Returns all boats & information on which ones are reserved.
### Full Outer Join

**Full Outer Join** returns all (matched or unmatched) rows from the tables on both sides of the join clause.

```
SELECT r.sid, b.bid, b.name
FROM Reserves r FULL OUTER JOIN Boats b
ON r.bid = b.bid
```

Returns all boats & all information on reservations.

### Views

**CREATE VIEW**

```
CREATE VIEW view_name
AS select_statement
```

- Makes development simpler
- Often used for security
- Not instantiated - makes updates tricky

**CREATE VIEW Reds**

```
CREATE VIEW Reds
AS SELECT B.bid, COUNT(*) AS scount
FROM Boats B, Reserves R
WHERE B.bid = R.bid AND B.color = 'red'
GROUP BY B.bid
```

```
b.bid scount
102 1
104 1
```

Note: in this case it is the same as the ROJ because bid is a foreign key in reserves, so all reservations must have a corresponding tuple in boats.