Life is just a bowl of queries.
-Anon
(not Forrest Gump)

SQL: The Query Language
Part 1
R&G - Chapter 5

The SQL Query Language
- The most widely used relational query language.
  - Current standard is SQL-1999
    - Not fully supported yet
    - Introduced “Object-Relational” concepts (and lots more)
      - Many of which were pioneered in PostgreSQL here at Berkeley!
    - SQL-200x is in draft
    - SQL-92 is a basic subset
      - Most systems support a medium
    - Postgres has some "unique" aspects
      - as do most systems.
    - XML support/integration is the next challenge for SQL
      (more on this in a later class).

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 re-order/change operations, and 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
- CREATE TABLE table_name
  ( ( column_name data_type [ DEFAULT default_expr ] [ column_constraint[,...] ] | table_constraint [,...] ) )

Create Table (w/column constraints)
- CREATE TABLE table_name
  ( ( column_name data_type [ DEFAULT default_expr ] [ column_constraint[,...] ] | table_constraint [,...] ) )

Create Table (w/table constraints)
- CREATE TABLE table_name
  ( ( column_name data_type [ DEFAULT default_expr ] [ column_constraint[,...] ] | table_constraint [,...] ) )

Table Constraints:
- [ CONSTRAINT constraint_name ]
  [ NOT NULL | NULL | UNIQUE ]
  | PRIMARY KEY | CHECK (expression)
  | REFERENCES reftable [ ( refcolumn ) ] [ ON DELETE action ] [ ON UPDATE action ]

For each column constraint must produce a boolean result and reference the related column’s value only.

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

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

Table Constraints:
- [ CONSTRAINT constraint_name ]
  [ NOT NULL | NULL | UNIQUE ]
  | PRIMARY KEY [ column_name [,...] ] | CHECK (expression)
  | REFERENCES reftable [ ( refcolumn [,...] ) ] [ ON DELETE action ] [ ON UPDATE action ]

Here, expressions, keys, 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 like ' % ') 
)

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'
```

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Carnatic101</td>
<td>C</td>
<td>3666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53668</td>
<td>Reggae203</td>
<td>B</td>
<td>3668</td>
<td>Smith</td>
<td>smith@cs</td>
<td>18</td>
<td>3.2</td>
</tr>
</tbody>
</table>

result =

<table>
<thead>
<tr>
<th>S.name</th>
<th>E.cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones</td>
<td>History105</td>
</tr>
</tbody>
</table>

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

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. (called "selection").
  3. do SELECT clause: Delete unwanted fields. (called "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.

Step 1 – Cross Product

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

<table>
<thead>
<tr>
<th>sid</th>
<th>S.name</th>
<th>S.login</th>
<th>E.cid</th>
<th>E.grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>53666</td>
<td>Carnatic101</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>53831</td>
<td>Reggae203</td>
</tr>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>53666</td>
<td>History105</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>53688</td>
<td>Carnatic101</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>53688</td>
<td>History105</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>53688</td>
<td>Topology112</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>53688</td>
<td>Topology112</td>
</tr>
</tbody>
</table>
Step 2) Discard tuples that fail predicate

<table>
<thead>
<tr>
<th>S.sid</th>
<th>S.name</th>
<th>S.hpnt</th>
<th>S.age</th>
<th>S.gender</th>
<th>E.sid</th>
<th>E.cid</th>
<th>E.grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>5</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>4.8</td>
</tr>
<tr>
<td>22</td>
<td>Dustin</td>
<td>5</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>4.8</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>5.5</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>5.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>3</td>
<td>63.5</td>
<td>3</td>
<td>102</td>
<td>3.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>3</td>
<td>63.5</td>
<td>3</td>
<td>102</td>
<td>3.5</td>
</tr>
</tbody>
</table>

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

Step 3) Discard Unwanted Columns

<table>
<thead>
<tr>
<th>S.sid</th>
<th>S.name</th>
<th>S.hpnt</th>
<th>S.age</th>
<th>S.gender</th>
<th>E.sid</th>
<th>E.cid</th>
<th>E.grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>5</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>4.8</td>
</tr>
<tr>
<td>22</td>
<td>Dustin</td>
<td>5</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>4.8</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>5.5</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>5.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>3</td>
<td>63.5</td>
<td>3</td>
<td>102</td>
<td>3.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>3</td>
<td>63.5</td>
<td>3</td>
<td>102</td>
<td>3.5</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>S.sid</th>
<th>S.name</th>
<th>S.hpnt</th>
<th>S.age</th>
<th>S.gender</th>
<th>E.sid</th>
<th>E.cid</th>
<th>E.grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>5</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>4.8</td>
</tr>
<tr>
<td>22</td>
<td>Dustin</td>
<td>5</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>4.8</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>5.5</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>5.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>3</td>
<td>63.5</td>
<td>3</td>
<td>102</td>
<td>3.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>3</td>
<td>63.5</td>
<td>3</td>
<td>102</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Example Schemas

CREATE TABLE Sailors (sid INTEGER PRIMARY KEY, sname CHAR(20), rating INTEGER, age REAL)
CREATE TABLE Boats (bid INTEGER PRIMARY KEY, bname CHAR (20), color CHAR(10))
CREATE TABLE Reserves (sid INTEGER REFERENCES Sailors, bid INTEGER, day DATE, PRIMARY KEY (sid, bid, day), FOREIGN KEY (bid) REFERENCES Boats)

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")

Can be rewritten using range variables as:

<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>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>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>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>3</td>
<td>102</td>
<td>10/10/96</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
<td>3</td>
<td>102</td>
<td>10/10/96</td>
</tr>
</tbody>
</table>

Another Join Query

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>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>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>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>3</td>
<td>102</td>
<td>10/10/96</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
<td>3</td>
<td>102</td>
<td>10/10/96</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")

Can be rewritten using range variables as:

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):

```
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:

```
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

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

- Can also have expressions in WHERE clause:

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

Find sailors who’ve reserved at least one boat

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

- Would adding DISTINCT to this query make a difference?
- What is the effect of replacing \$s.id\$ by \$s.name\$ in the SELECT clause?
  - Would adding DISTINCT to this variant of the query make a difference?

String operations

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

```
SELECT S.age, S.age-5 AS age1, 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.
- FYI -- this query doesn’t work in PostgreSQL!

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).

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

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

Find sid’s of sailors who’ve reserved a red and a green boat

- If we simply replace OR by AND in the previous query, we get the wrong answer. (Why?)
- Instead, could use a self-join:
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.
  - But PostgreSQL does!

---

Nested Queries

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

  **Names of sailors who’ve reserved boat #103:**

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

- 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:

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

- **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: op ANY, op ALL
- Find sailors whose rating is greater than that of some sailor called Horatio:

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

---

Rewriting INTERSECT Queries Using IN

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

```sql
SELECT R.sid
FROM Boats B, Reserves R
WHERE R.bid=B.bid 
  AND R.color=‘red’
AND R.sid IN (SELECT R2.sid
FROM Boats B2, Reserves R2
WHERE R2.bid=B2.bid 
  AND B2.color=‘green’)
```

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

---

Division in SQL

- *Find sailors who’ve reserved all boats*:

```sql
SELECT S.fname
FROM Sailors S, Reserves R
WHERE NOT EXISTS (SELECT B.bid
FROM Boats B
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*
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. Will be covered in subsequent lectures.

Aggregate Operators

• Significant extension of relational algebra.

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

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.
  • PostgreSQL seems to run it

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, ..., 10:
    SELECT MIN (S.age) FROM Sailors S
    WHERE S.rating = i