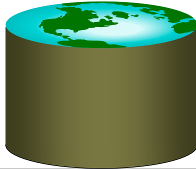
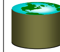
SQL: The Query Language Part 1

R&G - Chapter 5

Life is just a bowl of queries.

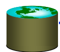
-Anon
(not Forrest Gump)





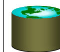
Relational Query Languages

- **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

- 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 Postgres here at Berkeley!
  – SQL-200x is in draft
  – SQL-92 is a basic subset
    - Most systems support a medium
  – PostgreSQL has some "unique" aspects
    - as do most systems.
  – XML support/integration is the next challenge for SQL (more on this in a later class).



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
  …
- **PostgreSQL also allows OIDs, arrays, inheritance, rules…**
  conformance to the SQL-1999 standard is variable so we won't use these in the project.

Create Table (w/column constraints)

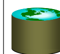
- **CREATE TABLE *table_name***
  ({ *column_name data_type* [DEFAULT *default_expr*] [*column_constraint* [, ...]] | *table_constraint* } [, ...])

Column Constraints:

- **[CONSTRAINT *constraint_name*]**
  { NOT NULL | NULL | UNIQUE | PRIMARY KEY | CHECK (*expression*) |
  REFERENCES *reftable* [(*refcolumn*)] [ON DELETE *action*] [ON UPDATE *action*] }

***action* is one of:**
NO ACTION, CASCADE, SET NULL, SET DEFAULT

***expression* for column constraint must produce a boolean result and reference the related column's value only.**



Create Table (w/table constraints)

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

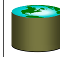
Table Constraints:

- **[CONSTRAINT *constraint_name*]**
  { UNIQUE (*column_name* [, ...]) |
  PRIMARY KEY (*column_name* [, ...]) |
  CHECK (*expression*) |
  FOREIGN KEY (*column_name* [, ...]) 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 <> ‘ ’)
);
```



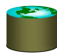
The SQL DML

- Single-table queries are straightforward.
- To find all 18 year old students, we can write:

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

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2

- 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'
```

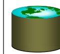
sid	cid	grade
53831	Carnatic101	C
53831	Reggae203	B
53650	Topology112	A
53666	History105	B

sid	name	login	age	gpa
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2

result =

S.name	E.cid
Jones	History105

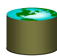
Note: obviously no referential integrity constraints have been used here.



Basic SQL Query

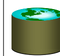
SELECT [DISTINCT] *target-list*
FROM *relation-list*
WHERE *qualification*

- *relation-list* : A list of relation names
  - possibly with a *range-variable* after each name
- *target-list* : A list of attributes of 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 $<, >, =, \leq, \geq, \neq$
- *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

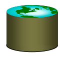
- Semantics of an SQL query are defined in terms of the following conceptual evaluation strategy:
  1. do FROM clause: compute *cross-product* of tables (e.g., Students and Enrolled).
  2. do WHERE clause: Check conditions, discard tuples that fail. (called "selection").
  3. do SELECT clause: Delete 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

S.sid	S.name	S.login	S.age	S.gpa	E.sid	E.cid	E.grade
53666	Jones	jones@cs	18	3.4	53831	Carnatic101	C
53666	Jones	jones@cs	18	3.4	53832	Reggae203	B
53666	Jones	jones@cs	18	3.4	53650	Topology112	A
53666	Jones	jones@cs	18	3.4	53666	History105	B
53688	Smith	smith@ee	18	3.2	53831	Carnatic101	C
53688	Smith	smith@ee	18	3.2	53831	Reggae203	B
53688	Smith	smith@ee	18	3.2	53650	Topology112	A
53688	Smith	smith@ee	18	3.2	53666	History105	B

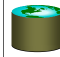
```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid AND E.grade='B'
```



Step 2) Discard tuples that fail predicate

S.sid	S.name	S.login	S.age	S.gpa	E.sid	E.cid	E.grade
53666	Jones	jones@cs	18	3.4	53831	Carnatic101	C
53666	Jones	jones@cs	18	3.4	53832	Reggae203	B
53666	Jones	jones@cs	18	3.4	53650	Topology112	A
53666	Jones	jones@cs	18	3.4	53666	History105	B
53688	Smith	smith@ee	18	3.2	53831	Carnatic101	C
53688	Smith	smith@ee	18	3.2	53831	Reggae203	B
53688	Smith	smith@ee	18	3.2	53650	Topology112	A
53688	Smith	smith@ee	18	3.2	53666	History105	B

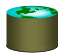
```
SELECT S.name, E.cid
 FROM Students S, Enrolled E
 WHERE S.sid=E.sid AND E.grade='B'
```



Step 3) Discard Unwanted Columns

S.sid	S.name	S.login	S.age	S.gpa	E.sid	E.cid	E.grade
53666	Jones	jones@cs	18	3.4	53831	Carnatic101	C
53666	Jones	jones@cs	18	3.4	53832	Reggae203	B
53666	Jones	jones@cs	18	3.4	53650	Topology112	A
53666	Jones	jones@cs	18	3.4	53666	History105	B
53688	Smith	smith@ee	18	3.2	53831	Carnatic101	C
53688	Smith	smith@ee	18	3.2	53831	Reggae203	B
53688	Smith	smith@ee	18	3.2	53650	Topology112	A
53688	Smith	smith@ee	18	3.2	53666	History105	B

```
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.

(Question: If the key for the Reserves relation contained only the attributes *sid* and *bid,* how would the semantics differ?)

Reserves

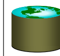
sid	bid	day
22	101	10/10/96
95	103	11/12/96

Sailors

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
95	Bob	3	63.5

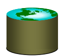
Boats

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red



Example Schemas

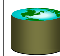
```
CREATE TABLE Sailors (sid INTEGER PRIMARY KEY,
   sname CHAR(20),rating INTEGER,age REAL)

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)
```



Another Join Query

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

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
95	Bob	3	63.5	22	101	10/10/96
95	Bob	3	63.5	95	103	11/12/96



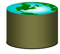
Some Notes on Range Variables

- 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")

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

Can be rewritten using range variables as:

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



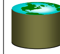
More Notes

- 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:

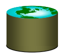
```
SELECT  *
FROM Sailors x
WHERE  x.age > 20
```



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.sid* by *S.sname* in the SELECT clause?
  - Would adding DISTINCT to this variant of the query make a difference?



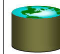
Expressions

- 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
```



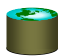
String operations

- 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!



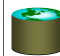
Find sid's of sailors who've reserved a red **or** a green boat

- 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')
```

Vs.

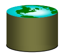
```
SELECT  R.sid
  FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='red'
UNION
SELECT R.sid
  FROM Boats B, Reserves R
WHERE R.bid=B.bid AND B.color='green'
```



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:

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

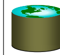


AND Continued…

- 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!

Key field!

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



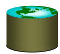
Nested Queries

- **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:

```
SELECT  S.sname
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.*

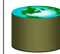


Nested Queries with Correlation

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

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

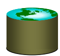
- **EXISTS is another set comparison operator, like IN.**
- **Can also specify NOT EXISTS**
- **If UNIQUE is used, and * is replaced by *R.bid*, finds sailors with at most one reservation for boat #103.**
  - UNIQUE checks for duplicate tuples in a subquery;
- **Subquery must be recomputed for each Sailors 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:**

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

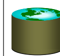


Rewriting INTERSECT Queries Using IN

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

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

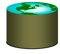
- **Similarly, EXCEPT queries re-written using NOT IN.**
- **How would you change this to find *names* (not *sid*'s) of Sailors who've reserved both red and green boats?**



Division in SQL

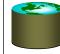
Find sailors who've reserved all boats.

```
SELECT S.sname
FROM  Sailors S                      Sailors S such that ...
WHERE  NOT EXISTS  (SELECT  B.bid    there is no boat B without
                    FROM  Boats B      ...
                    WHERE  NOT EXISTS  (SELECT  R.bid
a Reserves tuple showing S reserved B   FROM  Reserves R
                                        WHERE  R.bid=B.bid
                                          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. Will be covered in subsequent lectures.



Aggregate Operators

- **Significant extension of relational algebra.**

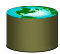
COUNT (*)
COUNT ([DISTINCT] A)
SUM ([DISTINCT] A)
AVG ([DISTINCT] A)
MAX (A)
MIN (A)

single column

```
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'
```



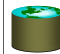
Aggregate Operators

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

single column

```
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
```



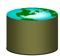
Find name and age of the oldest sailor(s)

- **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

```
SELECT S.sname, MAX (S.age)
FROM Sailors S
```

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

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



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
```