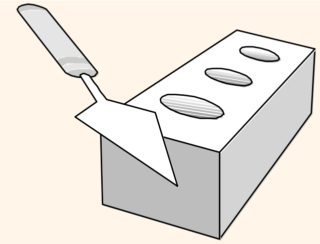
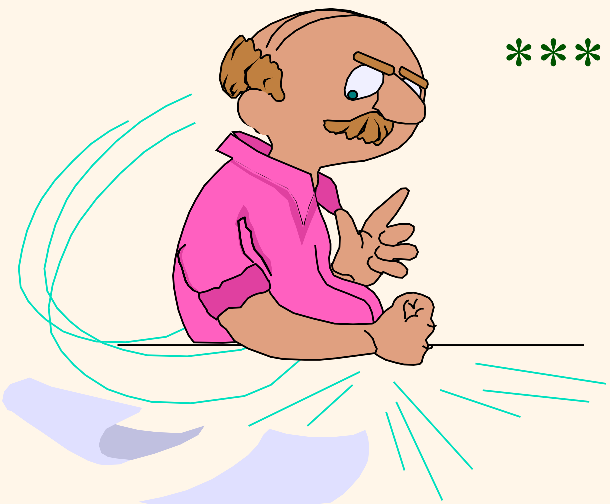


Introduction to Data Management

**** The "Flipped" Edition ****



Lecture #13
(SQL II)

Instructor: Mike Carey
mjcarey@ics.uci.edu

Today's Notices

- Midterm #1 is behind you!
  - Not a technical train wreck! (!)
  - Put it out of your minds for awhile...
- HW notes:
  - HW #2 is all graded (as you know)
  - HW #3's grading is currently in progress
  - HW #4 is your current entertainment – and we will now be on a "Friday pattern" for the next series of HW release dates and due dates
- Let's have a look at where are now, in terms of the course material...

Post-Midterm Roadmap Check

Topic Coverage and Exam Schedule

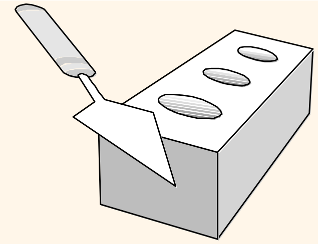
Syllabus

Topic	Reading (Required!)
~~Databases and DB Systems~~	Ch. 1
~~Entity-Relationship (E-R) Data Model~~	Ch. 6.1-6.5, 6.8-6.9
~~Relational Data Model~~	Ch. 2.1-2.4, 3.1-3.2
~~E-R to Relational Translation~~	Ch. 6.6-6.7
~~Relational Design Theory~~	Ch. 7.1-7.4.2
~~***Midterm Exam 1***~~	***Fri, Oct 22*** (during lecture time)
Relational Algebra	Ch. 2.5-2.7
Relational Calculus	Wikipedia: Tuple relational calculus
SQL Basics (SPJ and Nested Queries)	Ch. 3.3-3.5
SQL Analytics: Aggregation, Nulls, and Outer Joins	Ch. 3.6-3.9, 4.1
Advanced SQL: Constraints, Triggers, Views, and Security	Ch. 4.2, 4.4-4.5, 4.7
Midterm Exam 2	***Mon, Nov 15*** (during lecture time)
Storage	Ch. 12.1-12.4, 12.6-12.7
Indexing	Ch. 14.1-14.4, 14.5
Physical DB Design	Ch. 14.6-14.7, 15.1-15.3, 15.5.3
Semistructured Data Management (*a.k.a.* NoSQL)	Ch. 8.1, AsterixDB SQL++ Primer, Couchbase SQL++ Book
Data Science 1: Advanced SQL Analytics	Ch. 5.5, 11.3
Data Science 2: Notebooks, Dataframes, and Python/Pandas	Lecture notes and Jupyter notebook
Basics of Transactions	Ch. 4.3, Ch. 17
Endterm Exam	***Fri, Dec 3*** (during lecture time)

~~Midterm Exam 1~~

~~Time: Fri, Oct 22, Lecture Time~~
~~Place: SSLH 100~~

Nested Queries in SQL



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 clause can itself contain an SQL query! (Actually, so can SQL's FROM and HAVING clauses!)
- To find sailors who've *not* reserved #103, use NOT IN.
- To understand semantics (including **cardinality**) of nested queries, think *nested loops* evaluation: *For each Sailors tuple, check qualification by computing subquery.*

Nested Queries with Correlation

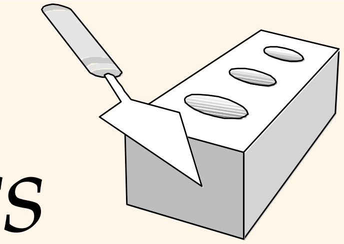
Find names of sailors who've reserved boat #103:

```
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, a subquery must be re-computed for each Sailors tuple (conceptually).

NOTE: Recall that there was a join way to express this query, too. Relational query optimizers will try to unnest queries into joins when possible to avoid nested loop query evaluation plans.

More on Set-Comparison Operators



- We've already seen IN and EXISTS.. Can also use NOT IN and NOT EXISTS.
- Also available: *op* ANY, *op* ALL (for *ops:* $<, >, \leq, \geq, =, \neq$)
- 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')
```

> *So let's try …*
> … running w/ANY on PostgreSQL
> … running w/ALL on PostgreSQL

Rewriting INTERSECT Queries Using IN

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

```
SELECT  S.sid
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
          AND S.sid IN  (SELECT  S2.sid
                         FROM  Sailors S2, Boats B2, Reserves R2
                         WHERE  S2.sid=R2.sid AND R2.bid=B2.bid
                                AND  B2.color='green')
```

- Similarly, EXCEPT queries can be re-written using NOT IN.
- This is what you'll have to do if using MySQL (but all the set ops are available in PostgreSQL ☺).

Division, SQL Style

Find sailors who've reserved **all** boats.

(1)

```
SELECT  S.sname
FROM  Sailors S
WHERE  NOT EXISTS
   ((SELECT  B.bid
     FROM  Boats B)
   EXCEPT
    (SELECT  R.bid
     FROM  Reserves R
     WHERE  R.sid=S.sid))
```

(This Sailor's unreserved Boat ids..!.)

Sailors S such that ...

*the set of **all** Boat ids ...*

***minus** ...*

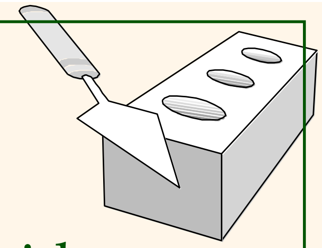
***this** Sailor's reserved Boat ids...*

*is **empty!***

Division in SQL (cont.)

(1)

```
SELECT  S.sname
FROM  Sailors S
WHERE  NOT EXISTS
        ((SELECT  B.bid
          FROM  Boats B)
         EXCEPT
          (SELECT  R.bid
           FROM  Reserves R
           WHERE  R.sid=S.sid))
```



Find sailors who've reserved all boats.

- Let's do it the hard(er) way, i.e.., without EXCEPT:

(2)

```
SELECT  S.sname
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 saying that **S** reserved B*

*This way is **not** that **non**-easy to understand – right...?* (☺)

Ordering and/or Limiting Query Results

Find the ratings, ids, names, and ages of the three best sailors

```
SELECT  S.rating, S.sid, S.sname, S.age
FROM  Sailors S
ORDER BY S.rating DESC
LIMIT 3
```

- The general syntax for this:

```
SELECT [DISTINCT] expressions
FROM tables
[WHERE condition]
  ....
[ORDER BY expression [ ASC | DESC ]]
LIMIT number_rows [ OFFSET offset_value ];
```

Aggregate Operators

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

single column

- Significant extension of the relational algebra.

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

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

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

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

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

(☺?)

Find name and age of the oldest sailor(s)

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

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

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

- That first try is *illegal*! (You'll see why shortly, when we do GROUP BY.)
- *Nit:* The third version is equivalent to the second one, and is allowed in the SQL/92 standard, but not supported in some early systems.

Motivation for Grouping

- 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
```

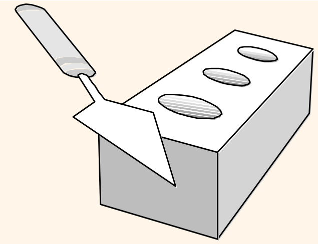
Queries With GROUP BY *and* HAVING

```
SELECT      [DISTINCT]  target-list
FROM        relation-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
```

- The *target-list* contains (i) attribute names and (ii) terms with aggregate operations (e.g., MIN (*S.age*)).
  - The attribute list (i) must be a subset of *grouping-list*. Intuitively, each answer tuple corresponds to a *group*, and these attributes must have **a single value per group**. (A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.)

Conceptual Evaluation

- The cross-product of *relation-list* is computed, tuples that fail the *qualification* are discarded, "*unnecessary*" fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in *grouping-list*.
- A *group-qualification* (HAVING) is then applied to eliminate some groups. Expressions in *group-qualification* must also have a *single value per group*!
  - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op must appear in *grouping-list*. (*But:* Some systems consider primary key semantics here.)
- One answer tuple is generated per qualifying group.



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

```
SELECT  S.rating,  MIN(S.age)
                   AS minage
FROM  Sailors S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING  COUNT(*) >= 2
```

Sailors instance:

sid	sname	rating	age
22	dustin	7	45.0
29	brutus	1	33.0
31	lubber	8	55.5
32	andy	8	25.5
58	rusty	10	35.0
64	horatio	7	35.0
71	zorba	10	16.0
74	horatio	9	35.0
85	art	3	25.5
95	bob	3	63.5
96	frodo	3	25.5

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

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

rating	age
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
10	16.0
9	35.0
3	25.5
3	63.5
3	25.5

rating	age
1	33.0
3	25.5
3	63.5
3	25.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

rating	minage
3	25.5
7	35.0
8	25.5

To Be Continued…