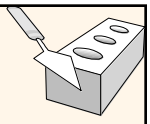
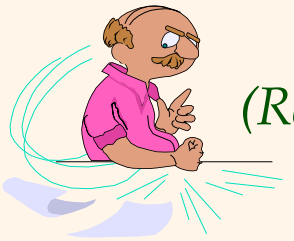
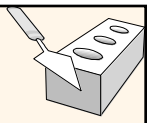


Introduction to Data Management



Lecture #14
(Relational Languages IV)

Instructor: Mike Carey
mjcarey@ics.uci.edu

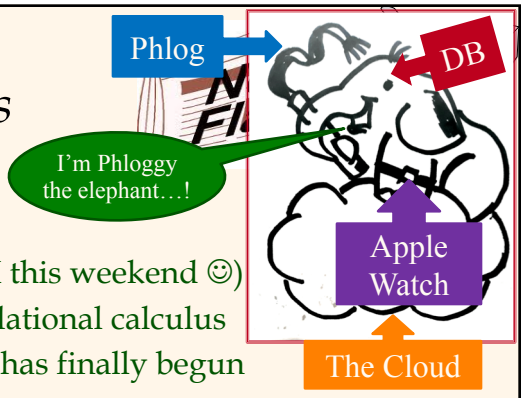


It's time again for....

Friday Nights with Databases

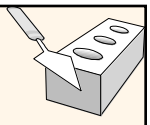
Brought to you by…

Announcements



- HW#4 is in flight...!
  - Due Monday (RelaX this weekend ☺)
  - Finished with the relational calculus
  - Our SQL adventure has finally begun
- HW#5 due out Monday ("*Monday mode*" for now)
  - First of a series of SQL-based HW assignments
  - Critical that you resolve any MySQL issues! (Take your machine to discussion section, post questions on Piazza, whatever it takes – else you won't survive…!)
  - ***We now have a project logo…*** ☺

Example Data in MySQL



Sailors

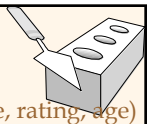
sid	sname	rating	age
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	4	25.5
95	Bob	3	63.5
101	Joan	3	NULL
107	Johan…	NULL	35.0

Reserves

sid	bid	date
22	101	1998-10-10
22	102	1998-10-10
22	103	1998-10-08
22	104	1998-10-07
31	102	1998-11-10
31	103	1998-11-06
31	104	1998-11-12
64	101	1998-09-05
64	102	1998-09-08
74	103	1998-09-08
NULL	103	1998-09-09
1	NULL	2001-01-11
1	NULL	2002-02-02

Boats

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



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

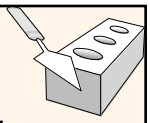
Sailors(sid, sname, rating, age)
Reserves(sid, bid, day)
Boats(bid, bname, color)

- If we replace OR by AND in this first version, what do we get?
- UNION: Can be used to compute the union of any two *union-compatible* sets of tuples (which are themselves the result of SQL queries).
- Also available: EXCEPT (What would we get if we replaced UNION by EXCEPT?)

[Note: MySQL vs. RelaX – and why?]

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

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

SQL vs. TRC

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

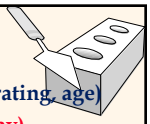
Sailors(sid, sname, rating, age)
Reserves(sid, bid, day)
Boats(bid, bname, color)

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

$$\{ t(sid) \mid \exists s \in Sailors\ (t.sid = s.sid \wedge \exists r \in Reserves\ (r.sid = s.sid \wedge \exists b \in Boats\ (b.bid = r.bid \wedge (b.color = \text{'red'} \vee b.color = \text{'green'})))) \}$$

这个需要一直括号里面这样去链接两个table

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



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

- INTERSECT: Can be used to compute the intersection of any two *union-compatible* sets of tuples.
- Included in the SQL/92 standard, but not in all systems (incl. MySQL).
- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

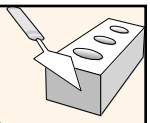
```
SELECT  S.sid
FROM  Sailors S, Boats B1, Reserves R1,
        Boats B2, Reserves R2
WHERE  S.sid=R1.sid AND R1.bid=B1.bid
  AND  S.sid=R2.sid AND R2.bid=B2.bid
  AND (B1.color= 'red' AND B2.color= 'green' )
```

这里用了两次一样的两个东西，所以可以认为是一个在找红色的，另一个在找绿色的

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



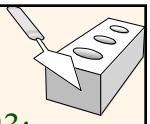
Find names of sailors who've reserved boat #103:

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

WHere, from, having 都可以用in

- 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



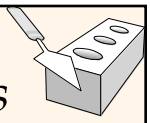
Find names of sailors who've reserved boat #103:

```
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, 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

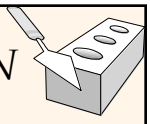


- We've already seen IN and EXISTS.. Can also use NOT IN and NOT EXISTS.
- Also available: *op* ANY, *op* ALL (for *ops:* <, >, ≤, ≥,=, ≠)
- Find sailors whose rating is greater than that of some sailor called Horatio:

So let's try ...
… running w/ANY on MySQL
… running w/ALL on MySQL

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

Rewriting INTERSECT Queries Using IN

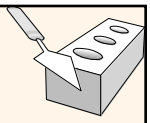


Find sid's of sailors who've reserved both a red and a green boat:

```
SELECT 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 need to do when using MySQL (but you can play with RelaX for the other set ops).

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))
```

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

Sailors S such that …

***there is no** boat B **without** …*

a Reserves tuple saying that S reserved B

Ordering and/or Limiting Query Results

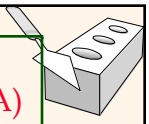
Find the ratings, ids, names, and ages of the three best sailors

```
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
```

总共有多少条

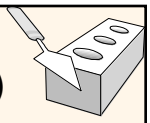
```
SELECT  AVG(S.age)
FROM  Sailors S
WHERE  S.rating=10
```

```
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)



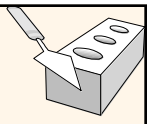
~~SELECT S.sname, MAX (S.age)~~
~~FROM Sailors S~~

- That first try is *illegal*! (We'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 all systems.

```
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
```

Motivation for Grouping

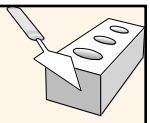


- So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.
- Consider: *Find the age of the youngest sailor for each rating level.*
  - In general, we don't know how many rating levels exist, and what the rating values 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:

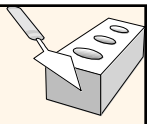
```
SELECT  MIN(S.age)
FROM  Sailors S
WHERE  S.rating = i
```


Queries With GROUP BY and HAVING



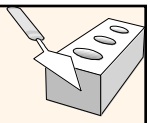
```
SELECT      [DISTINCT] target-list
FROM        relation-list
WHERE       qualification
GROUP BY    grouping-list
HAVING      group-qualification
```

- 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*. (Note: SQL doesn't 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.

每个group里面的age的平均数

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

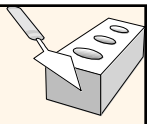
Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

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

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