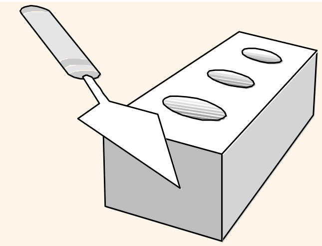
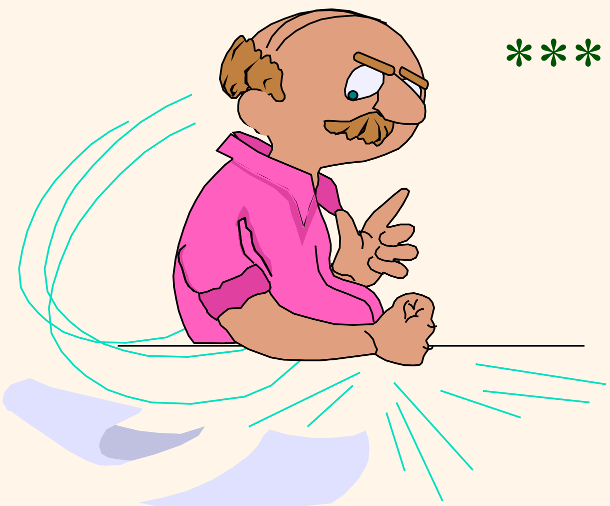


Introduction to Data Management

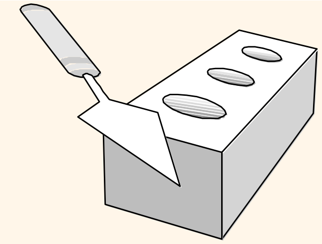
**** The "Flipped" Edition ****



Lecture #12
(SQL I)

Instructor: Mike Carey
mjcarey@ics.uci.edu

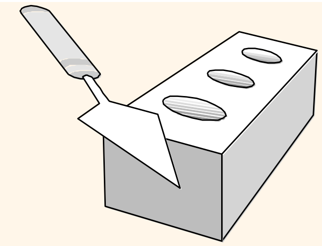
Today's Notices



- *SWOOSH* HW series status
  - HW3 is winding down! (Due Wed at 6 PM)
  - HW4 will be out on Friday (not Wed!) this week

- Midterm 1 is this Friday!
  - See Piazza for the full set of rules
    - ***In-person*** exam (but Gradescope)
    - Laptops needed (open ***only*** to Gradescope!)
    - 2-sided ***hardcopy*** cheat sheet permitted
    - Assigned seating – be sure to ***come early!***

- Today our SQL adventure begins...
  - Watch the "SQL Pre-Lecture" before watching this!



Pre-Midterm Time Check!

Topic Coverage and Exam Schedule

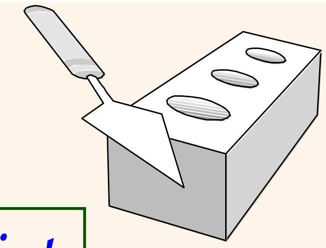
Syllabus

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

On to ***SQL****…!*



SQL "SPJ" Query:

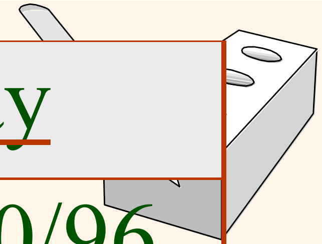
```
SELECT    [DISTINCT]  target-list
FROM      relation-list
WHERE     qualification
```

- *relation-list* A list of relation names (possibly with a *range-variable* after each name).
- *target-list* A list of attributes of relations in *relation-list*
- *qualification* Comparisons (Attr *op* const or Attr1 *op* Attr2, where *op* is one of <, <=, =, >, >=, <>) combined using AND, OR and NOT.
- DISTINCT is an optional keyword indicating that the answer should not contain duplicates. Default is that duplicates are *not* eliminated! (***Bags***, not sets.)

Many SQL-Based DBMSs

- Commercial RDBMS choices include
  - DB2 (IBM)
  - Oracle
  - SQL Server (Microsoft)
  - Teradata
- Open source RDBMS options include
  - MySQL
  - PostgreSQL
- And for so-called "Big Data", we also have
  - Apache Hive (on Hadoop/Tez), Spark SQL, ...

Example Instances



- We'll use these instances of our usual Sailors and Reserves relations in our examples again.

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S2

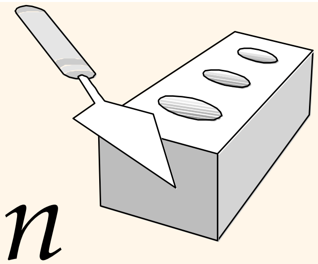
sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Conceptual Evaluation Strategy

- Semantics of an SQL SPJ query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of *relation-list*. ($\times$)
  - Discard resulting tuples if they fail *qualifications*. (σ)
  - Project out attributes that are not in *target-list*. (π)
  - If DISTINCT is specified, eliminate duplicate rows. (δ)

- This strategy is probably the ***least*** efficient way to compute a query! An optimizer will find more efficient strategies to compute *the same answers*.

Example of Conceptual Evaluation



Let's go check out ***PostgreSQL***...!

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

← *using S1 for Sailors*

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	$\boxed{103}$	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	$\boxed{103}$	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	$\boxed{103}$	11/12/96

A Note on Range Variables

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

- Named variables "needed" when the same relation appears twice (or more) in the FROM clause. Previous query can be written lazily:

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

OR

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

It's better style, though, to use range variables – ***always...!***

Example Data in PostgreSQL

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

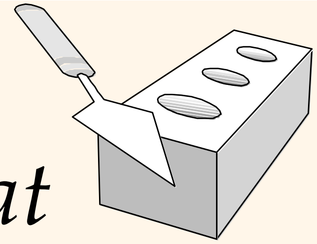
Reserves

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 sailors who've reserved at least one boat



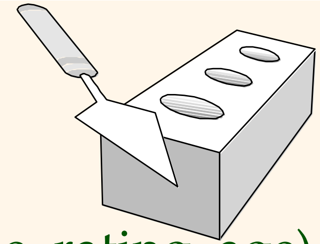
Sailors(sid, sname, rating, age)
Reserves(sid, bid, day)
Boats(bid, bname, color)

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

Let's go back
to ***PostgreSQL***...!

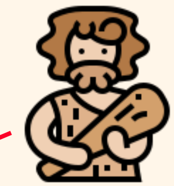
- Would adding DISTINCT to this query make a difference? (With our example data? And what about other possible data?)
- What is the effect of replacing *S.sid* by *S.sname* in the SELECT clause? Would adding DISTINCT to *this* variant of the query make a difference?

Expressions and Strings



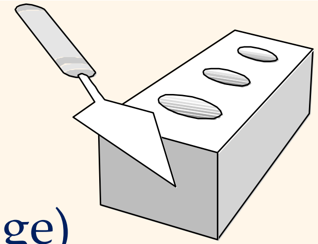
Sailors(sid, sname, rating, age)

```
SELECT  S.sname, S.age, S.age/7.0 AS dogyears
FROM  Sailors S
WHERE  S.sname LIKE 'Z_%a'
```



- Illustrates use of arithmetic expressions and string pattern matching: *Find triples (names and ages of sailors plus a field defined by an expression) for sailors whose names begin and end with B and contain at least three characters.*
- AS provides a way to (re)name fields in result.
- LIKE is used for string matching. `_' stands for any one character and `%' stands for 0 or more arbitrary characters. (See PostgreSQL docs for more info...)

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



Sailors(sid, sname, rating, age)

Reserves(sid, bid, date)

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?)

```
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')
```

[Note: PostgreSQL vs. MySQL – and why?]

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, date)
Boats(bid, bname, color)

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

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

SQL vs. TRC (Take 2):

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

Sailors(sid, sname, rating, age)

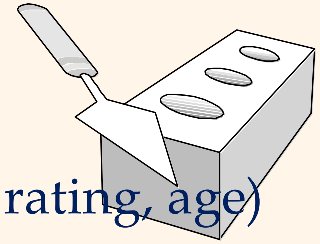
Reserves(sid, bid, date)

Boats(bid, bname, color)

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

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

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



Sailors(sid, sname, rating, age)

Reserves(sid, bid, date)

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 two 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')
```

Key field!* *(Q: why?)

```
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'
```

To Be Continued...