Database Management Systems
Session 5

Instructor: Vinnie Costa
vcosta@optonline.net

Term Paper

- Due Saturday, Oct 8
- Should be about 3-4 pages (9 or 10 font)
- Some people still have not submitted topics

Homework

- Read Chapter Three
- No exercises for next class; MidTerm instead
- Any Questions?

MidTerm Exam

- Due **today**, September 17
- No late submissions

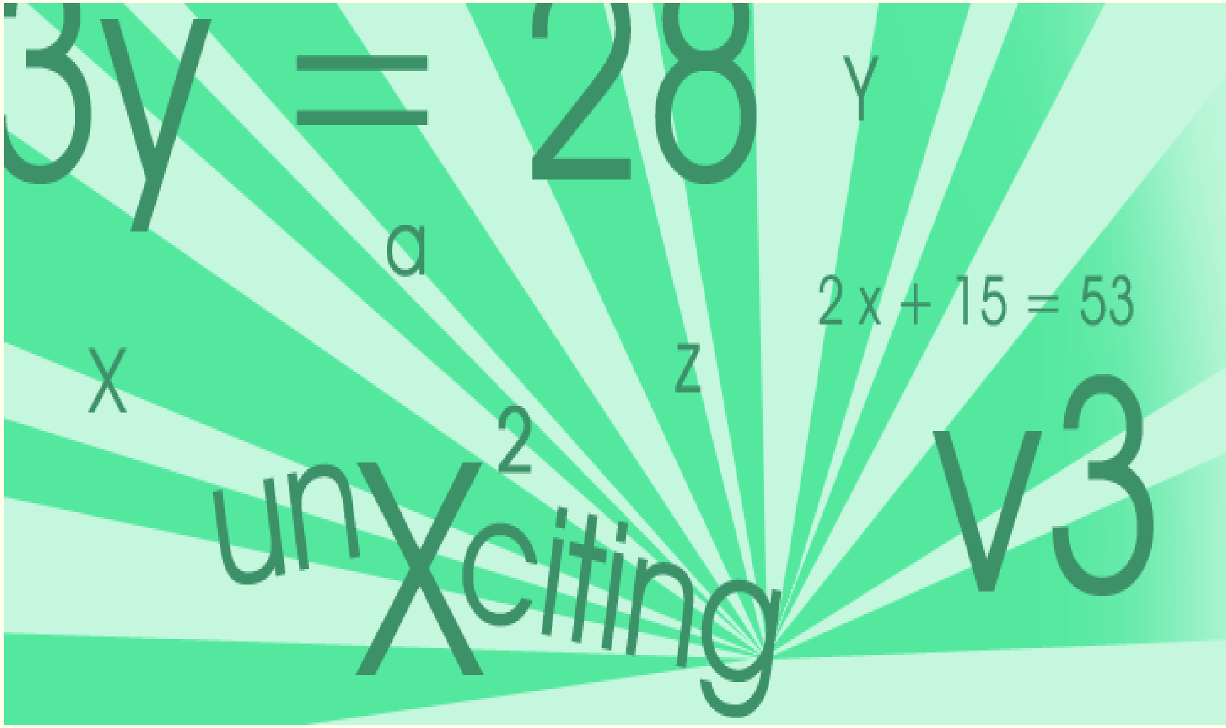
Homework

- Install PHP On Your System
- Install MySQL
- Create, Delete, Modify Tables
- Insert, Modify, Delete Data Into Tables
- Play with MySQL
- Any Trouble?

Oracle Buys Siebel

- September 12, 2005 – Oracle will acquire customer-service software specialist Siebel Systems in a deal worth $5.85 billion. “In a single step, Oracle becomes the No. 1 CRM applications company in the world,” said Oracle CEO Larry Ellison.
- Oracle was founded in 1977 by Larry Ellison who has a net worth of over $18 Billion, making him the 9th richest man in the world!

Relational Algebra



Chapter 4, Part A

Relational Query Languages

- **Query languages (QL)** - specialized languages to **manipulate** and **retrieve** data from a database
- Relational model supports simple, powerful QLs:
  - Strong formal foundation based on set theory and logic
  - Allows for much optimization
- Query Languages **are** programming languages!
  - QLs not intended to be used for complex calculations.
  - QLs support easy, efficient access to large data sets.
- In the summer of 1979, **Relational Software, Inc.** (now **Oracle Corporation**) introduced the first commercially available implementation of SQL (beat IBM to market by two years) by releasing their first commercial RDBMS

Formal Relational Query Languages

- Two mathematical Query Languages form the basis for "real" languages (e.g. SQL), and for implementation:
  - **Relational Algebra** - More operational, very useful for representing execution plans (**procedural**)
  - **Relational Calculus** - Lets users describe what they want, rather than how to compute it. (Non-operational, **declarative**)

Preliminaries

- A query is applied to *relation instances*, and the result of a query is also a relation instance.
  - *Schemas* of input relations for a query are fixed (but query will run regardless of instance!)
  - The schema for the *result* of a given query is also fixed! Determined by definition of query language constructs.
- Positional vs. named-field notation:
  - Positional notation easier for formal definitions, named-field notation more readable.
  - Both used in SQL

Example Instances

R1

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

- **Sailors** (S1, S2) and **Reserves** (R1) relations for our examples
- We'll use **positional** or **named field** notation, assume that names of fields in query results are `inherited' from names of fields in query input relations

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

Relational Algebra

σ **Sigma**
π **Pi**

- Basic operations:
  - **Selection** (σ) Selects a subset of rows from relation
  - **Projection** (π) Deletes unwanted columns from relation
  - **Cross-product** ($\times$) Allows us to combine two relations
  - **Set-difference** ($-$) Tuples in reln. 1, but not in reln. 2
  - **Union** ($\cup$) Tuples in reln. 1 and in reln. 2
- Additional operations:
  - Intersection, **join**, division, renaming: Not essential, but (very!) useful
- Since each operation returns a relation, operations can be **composed**! (Algebra is "closed")

Selection

sid	sname	rating	age
28	yuppy	9	35.0
58	rusty	10	35.0

$$\sigma_{rating>8}(S2)$$

- Selects rows that satisfy **selection condition**
- No duplicates in result! (Why?)
- **Schema** of result identical to schema of (only) input relation
- **Result** relation can be the **input** for another relational algebra operation! (**Operator composition**)

sname	rating
yuppy	9
rusty	10

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$

Projection

- Deletes attributes that are not in **projection list**
- **Schema** of result contains exactly the fields in the projection list, with the same names that they had in the (only) input relation
- Projection operator has to eliminate **duplicates** (Why?)
    - Note: real systems typically don't do duplicate elimination unless the user explicitly asks for it. (Why not?)

sname	rating
yuppy	9
lubber	8
guppy	5
rusty	10

$$\pi_{sname,rating}(S2)$$

age
35.0
55.5

$$\pi_{age}(S2)$$

Union, Intersection, Set-Difference

- All of these operations take two input relations, which must be **union-compatible**:
  - Same number of fields.
  - **Corresponding** fields have the same type.
- The **schema** of result is identical to schema of input

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

$S1 \cup S2$

sid	sname	rating	age
22	dustin	7	45.0

$S1 - S2$

sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

$S1 \cap S2$

ρ **Rho**

Cross-Product

- Each row of S1 is **paired** with each row of R1.
- **Result schema** has one field per field of S1 and R1, with field names `inherited' if possible.
  - *Conflict*: Both S1 and R1 have a field called *sid*.

(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
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

- **Renaming operator**: $\rho\ (C(1 \rightarrow sid1, 5 \rightarrow sid2),\ S1 \times R1)$

Joins

- **Condition Join**: $R \bowtie_c S = \sigma_c (R \times S)$

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

$$S1 \bowtie_{S1.sid < R1.sid} R1$$

- **Result schema** same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently

Joins

- **Equi-Join**: A special case of condition join where the condition *c* contains only ***equalities*.**

sid	sname	rating	age		bid	day
22	dustin	7	45.0		101	10/10/96
58	rusty	10	35.0		103	11/12/96

$$S1 \bowtie_{sid} R1$$

- **Result schema** similar to cross-product, but only one copy of fields for which equality is specified.
- **Natural Join**: Equijoin on *all* common fields.

Division

- Not supported as a primitive operator, but useful for expressing queries like:

  *Find sailors who have reserved **all** boats.*

- Let *A* have 2 fields, *x* and *y*; *B* have only field *y*:
  - *A/B* = $\left\{\langle x\rangle \mid \exists \langle x, y\rangle \in A \;\; \forall \langle y\rangle \in B\right\}$
  - i.e., ***A/B* contains all *x* tuples (sailors) such that for *every* *y* tuple (boat) in *B*, there is an *xy* tuple in *A*.**
  - *Or*: If the set of *y* values (boats) associated with an *x* value (sailor) in *A* contains all *y* values in *B*, the *x* value is in *A/B*.
- In general, *x* and *y* can be any lists of fields; *y* is the list of fields in *B*, and $x \cup y$ is the list of fields of *A*.

Examples of Division A/B

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

A

pno
p2

B1

pno
p2
p4

B2

pno
p1
p2
p4

B3

sno
s1
s2
s3
s4

A/B1

sno
s1
s4

A/B2

sno
s1

A/B3

Find names of sailors who've reserved boat #103

- Solution 1: $\pi_{sname}((\sigma_{bid=103}\text{Reserves}) \bowtie Sailors)$

- Solution 2: $\rho\ (Temp1,\ \sigma_{bid=103}\ \text{Reserves})$

  $\rho\ (Temp2,\ Temp1 \bowtie Sailors)$

  $\pi_{sname}(Temp2)$

- Solution 3: $\pi_{sname}(\sigma_{bid=103}(\text{Reserves} \bowtie Sailors))$

Find names of sailors who've reserved a red boat

- Information about boat color only available in Boats; so need an extra join:

$$\pi_{sname}((\sigma_{color='red'}Boats) \bowtie \text{Reserves} \bowtie Sailors)$$

- A more efficient solution:

$$\pi_{sname}(\pi_{sid}((\pi_{bid}\sigma_{color='red'}Boats) \bowtie \text{Res}) \bowtie Sailors)$$

A query optimizer can find this, given the first solution!

Find sailors who've reserved a red or a green boat

- Can identify all red or green boats, then find sailors who've reserved one of these boats:

$$\rho\ (Tempboats, (\sigma_{color='red' \vee color='green'}\ Boats))$$

$$\pi_{sname}(Tempboats \bowtie \text{Re}serves \bowtie Sailors)$$

- What happens if $\vee$ is replaced by $\wedge$ in this query?

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

- Previous approach won't work! Must identify sailors who've reserved red boats, sailors who've reserved green boats, then find the intersection (note that *sid* is a key for Sailors):

$$\rho\ (Tempred, \pi_{sid}((\sigma_{color='red'} Boats) \bowtie \text{Reserves}))$$

$$\rho\ (Tempgreen, \pi_{sid}((\sigma_{color='green'} Boats) \bowtie \text{Reserves}))$$

$$\pi_{sname}((Tempred \cap Tempgreen) \bowtie Sailors)$$

*Find the names of sailors who've reserved **all** boats*

- Uses **division**; schemas of the input relations to / must be carefully chosen:

$$\rho\ (Tempsids, (\pi_{sid,bid}\mathrm{Re}serves)\ /\ (\pi_{bid}Boats))$$

$$\pi_{sname}(Tempsids \bowtie Sailors)$$

Summary

- The relational model has **rigorously defined query languages** that are simple and powerful
- **Relational algebra** is more **operational**; useful as internal representation for query evaluation plans
- **Several ways of expressing a given query**; a **query optimizer** should choose the most efficient version.

Relational Calculus

- Comes in two flavors: *Tuple relational calculus* (TRC) and *Domain relational calculus* (DRC).
- Calculus has *variables, constants, comparison ops, logical connectives* and *quantifiers*.
  - **TRC** - Variables range over (i.e., get bound to) *tuples*.
  - **DRC** - Variables range over *domain elements* (= field values).
  - Both **TRC** and **DRC** are **simple subsets** of **first-order logic**.
- Expressions in the calculus are called **formulas**. An answer row is essentially an assignment of constants to variables that make the formula evaluate to **true**

Domain Relational Calculus

- **Query** has the form:

$$\left\{\left\langle x1,x2,\ldots,xn\right\rangle \mid p\left(\left\langle x1,x2,\ldots,xn\right\rangle\right)\right\}$$

- **Answer** includes all tuples $\left\langle x1,x2,\ldots,xn\right\rangle$ that make the *formula* $p\left(\left\langle x1,x2,\ldots,xn\right\rangle\right)$ be *true.*
- **Formula** is recursively defined, starting with simple **atomic formulas** (getting rows from relations or making comparisons of values), and building bigger and better formulas using the **logical connectives.**

Summary

- Relational calculus is **non-operational**, and users define queries in terms of what they want, not in terms of how to compute it. **(Declarative)**
- Algebra and safe calculus have same expressive power, leading to the notion of relational completeness.

SQL: Queries, Constraints, Triggers

Chapter 5

Example Instances

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

- We will use these instances of the **Sailors** and **Reserves** relations in our examples

Basic SQL Query

```
SELECT      [DISTINCT]   select-list
FROM        from-list
WHERE       qualification
```

- **select-list** - A list of attributes of relations in *select-list*
- **from-list** - A list of relation names (possibly with a **range-variable** after each name).
- **qualification** - Comparisons (`Attr` *`op`* `const` or `Attr1` *`op`* `Attr2`, where *op* is one of $<, >, =, \leq, \geq, \neq$) 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!

Conceptual Evaluation Strategy

- Semantics of an SQL query defined in terms of the following **conceptual evaluation strategy**:
  - **Compute the cross-product** of **from-list**
  - **Discard** resulting tuples if they fail **qualifications**
  - **Delete attributes** that are not in **select-list**
  - If **DISTINCT** is specified, **eliminate duplicate rows**
- 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

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

Text p.137

(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
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

A Note on Range Variables

- Really needed only if the same relation appears twice in the FROM clause. The previous query can also be written as:

```
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 is good style, however, to use range variables always!

$$\pi_{sname}((\sigma_{bid=103}\text{Reserves})\bowtie \textit{Sailors})$$

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?

$$\pi_{sname}(Sailors \bowtie Reserves)$$

Expressions and Strings

```
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM   Sailors S
WHERE  S.sname LIKE ‘B_%B’
```

- Illustrates use of arithmetic expressions and string pattern matching: *Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names begin and end with B and contain at least three characters*
- **AS** and **=** are two ways to name fields in result
- **LIKE** is used for pattern matching. `**_**’ stands for any one character and `**%**’ stands for 0 or more arbitrary characters
- `**Bob**’ is the only pattern match

*Find sid's of sailors who've reserved a red **or** 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' OR B.color='green')
```

- If we replace OR by AND in the first version, what do we get?

```
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 B.color='green')
```

- Same boat cannot have two colors. Always returns an **empty** answer set!

*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 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'
```

- This query says that we want the union of the set of sailors who have reserved red boats and the set of sailors who have reserved green boats

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

- **INTERSECT** - Can be used to compute the union of any two **union-compatible** sets of tuples

```
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'
```

- This query has a subtle bug if we select *sname* instead of *sid*. *Sname* is **not a key** and we have two Horatio's, each with a different color boat!

Find sid's of sailors who've reserved red boats ***but not*** *green boats*

- **EXCEPT** - Can be used to compute set-difference of any two **union-compatible** sets of tuples

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

- A very powerful feature of SQL: a WHERE clause can itself contain an SQL query! (Actually, so can FROM and HAVING clauses.)
- 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.*

Multiply Nested Queries

Find names of sailors who have reserved a red boat

```
SELECT S.sname
FROM   Sailors S
WHERE  S.sid IN   (SELECT R.sid
                   FROM   Reserves R
                   WHERE  R.bid IN ( SELECT B.bid
                                     FROM   Boats B
                                     WHERE  B.color = ‘red’)
```

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)
```

correlation

- **EXISTS** is another set comparison operator, like **IN**.
- If **UNIQUE** is used, and * is replaced by *R.bid*, it finds sailors with at most one reservation for boat #103. (**UNIQUE** checks for duplicate tuples; * denotes all attributes)
- In general, subquery must be re-computed for each Sailors tuple.

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`**, *op* **`IN`** where *op* is one of {>,<,=,≥,≤,≠}
- *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 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 re-written using **`NOT IN`**
- To find *names* (not *sid*'s) of Sailors who've reserved both red and green boats, just replace *S.sid* by *S.sname* in **`SELECT`** clause.

Division in SQL

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 ))
```

Division in SQL

Find sailors who've reserved all boats

- Let's do it the hard way, 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 row showing S reserved B

Aggregate Operators

- Significant extension of relational algebra

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

single column

Find the average age of sailors with a rating of 10

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

Count the number of sailors

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

Count the number of different sailor names

```
SELECT COUNT(DISTINCT S.sname)
FROM  Sailors S
```

Find the name and age of the oldest sailor

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

Find name and age of the oldest sailor(s)

- The first query is illegal! (We'll look into the reason a bit later, when we discuss `GROUP BY`)

```
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)
```

- The third query is equivalent to the second query, and is allowed in the SQL/92 standard, but is **not supported in some systems**

```
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) rows. Sometimes, we want to apply them to each of several **groups** of rows
- 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]  select-list
FROM       from-list
WHERE      qualification
GROUP BY   grouping-list
HAVING     group-qualification
```

- The **select-list** contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (*S.age*)).
  - The attribute list (i) must be a subset of **grouping-list**. Intuitively, each answer row 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 **from-list** is computed, rows that fail **qualification** are discarded, `*unnecessary*' fields are deleted, and the remaining rows are partitioned into groups by the value of attributes in **grouping-list**
- The **group-qualification** is then applied to eliminate some groups. Expressions in *group-qualification* must have a **single value per group**
  - In effect, an attribute in *group-qualification* that is not an argument of an aggregate op also appears in *grouping-list*.
- One answer row 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 (*) > 1
```

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

We apply the WHERE clause

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

eliminated unwanted columns

dups

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

sort table by groups

rating	minage
3	25.5
7	35.0
8	25.5

apply `HAVING` clause

Find age of the youngest sailor with age ≥18, for each rating with at least 2 such sailors and with every sailor under 60.

introduced in SQL:1999

```
HAVING  COUNT (*) > 1 AND EVERY (S.age <=60)
```

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

this group dropped

rating	minage
7	35.0
8	25.5

What is the result of changing **EVERY** to **ANY**?

Find age of the youngest sailor with age ≥18, for each rating with at least 2 sailors between 18 and 60.

```
SELECT  S.rating,  MIN (S.age)
                    AS minage
FROM  Sailors S
WHERE  S.age >= 18 AND S.age <= 60
GROUP BY  S.rating
HAVING  COUNT (*) > 1
```

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

this group
still has two
row that meet
qualification

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

Null Values

- Field values in a row are sometimes **unknown** (e.g., a rating has not been assigned) or **inapplicable** (e.g., no spouse's name).
  - SQL provides a special value **null** for such situations.
- The presence of **null complicates** many issues. e.g.:
  - Special operators needed to check if value is/is not *null*.
  - Is ***`rating>8`*** **true** or **false** when *rating* is equal to *null*? What about AND, OR and NOT connectives?
  - We need a **3-valued logic** (true, false and **unknown**).
  - Meaning of constructs must be defined carefully. (e.g., WHERE clause eliminates rows that don't evaluate to true.)
  - New operators (in particular, *outer joins*) possible/needed.

Triggers

- **Trigger** - procedure that starts automatically if specified changes occur to the DBMS
- Three parts:
  - **Event** (activates the trigger)
  - **Condition** (tests whether the triggers should run)
  - **Action** (what happens if the trigger runs)

Triggers: Example (SQL:1999)

```
CREATE TRIGGER youngSailorUpdate
   AFTER INSERT ON Sailors
REFERENCING NEW TABLE NewSailors
FOR EACH STATEMENT
   INSERT
       INTO YoungSailors(sid, name, age, rating)
       SELECT sid, name, age, rating
       FROM NewSailors N
       WHERE N.age <= 18
```

Summary

- **SQL** was an important factor in the early **acceptance** of the **relational model**; more natural than earlier, procedural query languages
- **Relationally complete**; in fact, significantly **more expressive power than relational algebra**
- Even queries that can be expressed in RA can often be **expressed more naturally in SQL**
- Many **alternative ways to write a query**; **optimizer** should look for most **efficient** evaluation plan.
  - In practice, users need to be aware of how queries are optimized and evaluated for best results.

Summary (Contd.)

- **NULL** for unknown field values brings many complications
- **Triggers** respond to changes in the database

Homework

- Read Chapters Four and Five
- Only study topics covered in class