SQL

Ramakrishnan & Gehrke, Chapters 4 & 5

Example Instances

Sailors

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

Reserves

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

Boats

bid	color
101	red
102	blue
103	green

Basic SQL Query Structure

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

- relation-list
  - list of relation names (possibly with a range-variable after each name)
- target-list
  - A list of attributes of relations in relation-list, possibly using range variables
- qualification
  - *Attr op const* or *Attr1 op Attr2* where op one of $<, >, =, \neq, \leq, \geq$ combined using AND, OR, NOT
- DISTINCT is optional for suppressing duplicates
  - By default duplicates not eliminated! …so tables actually are multisets, not sets

Conceptual Evaluation Strategy

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

- Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of relation-list
  - Discard resulting tuples if they fail qualification
  - Delete attributes that are not in target-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

JACOBS UNIVERSITY

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

(sid)	sname	rating	age	(sid)	bid	day
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	1	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	7	35.0	58	103	11/12/96

- cardinality?

A Note on Range Variables

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

```
SELECT  S.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!

Join

- Join = several tables addressed in one query

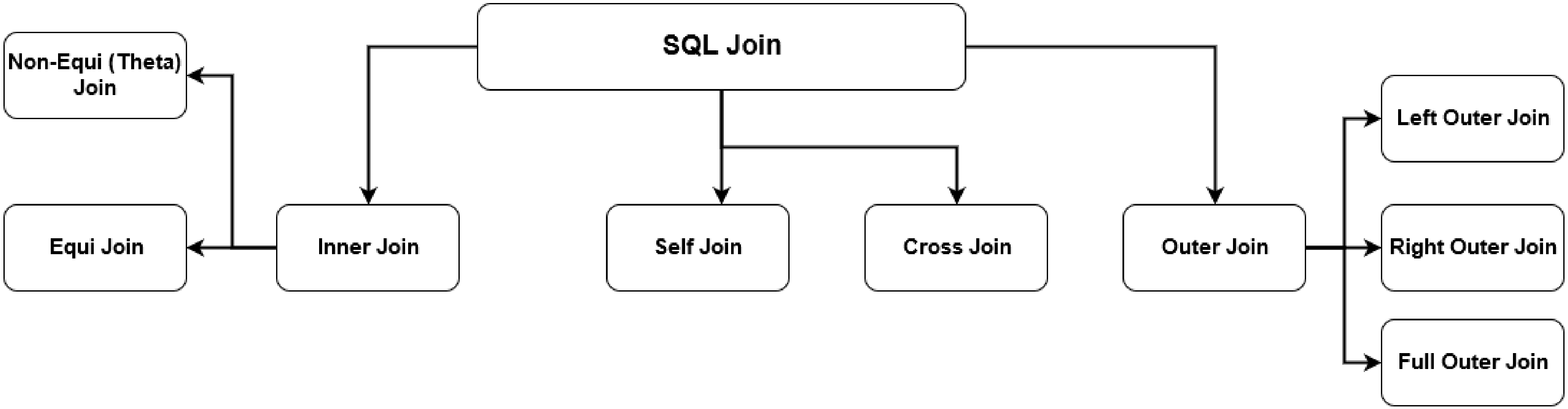
```
SELECT  target-list
FROM  Relation1 R1, Relation2 R2, …
WHERE  qualification
```

- List of relations in FROM clause determine cross product
- Frequently cross-relation conditions on attribute values to restrict results
- Most common: R1.attr1 = R2.attr2
  - ex:

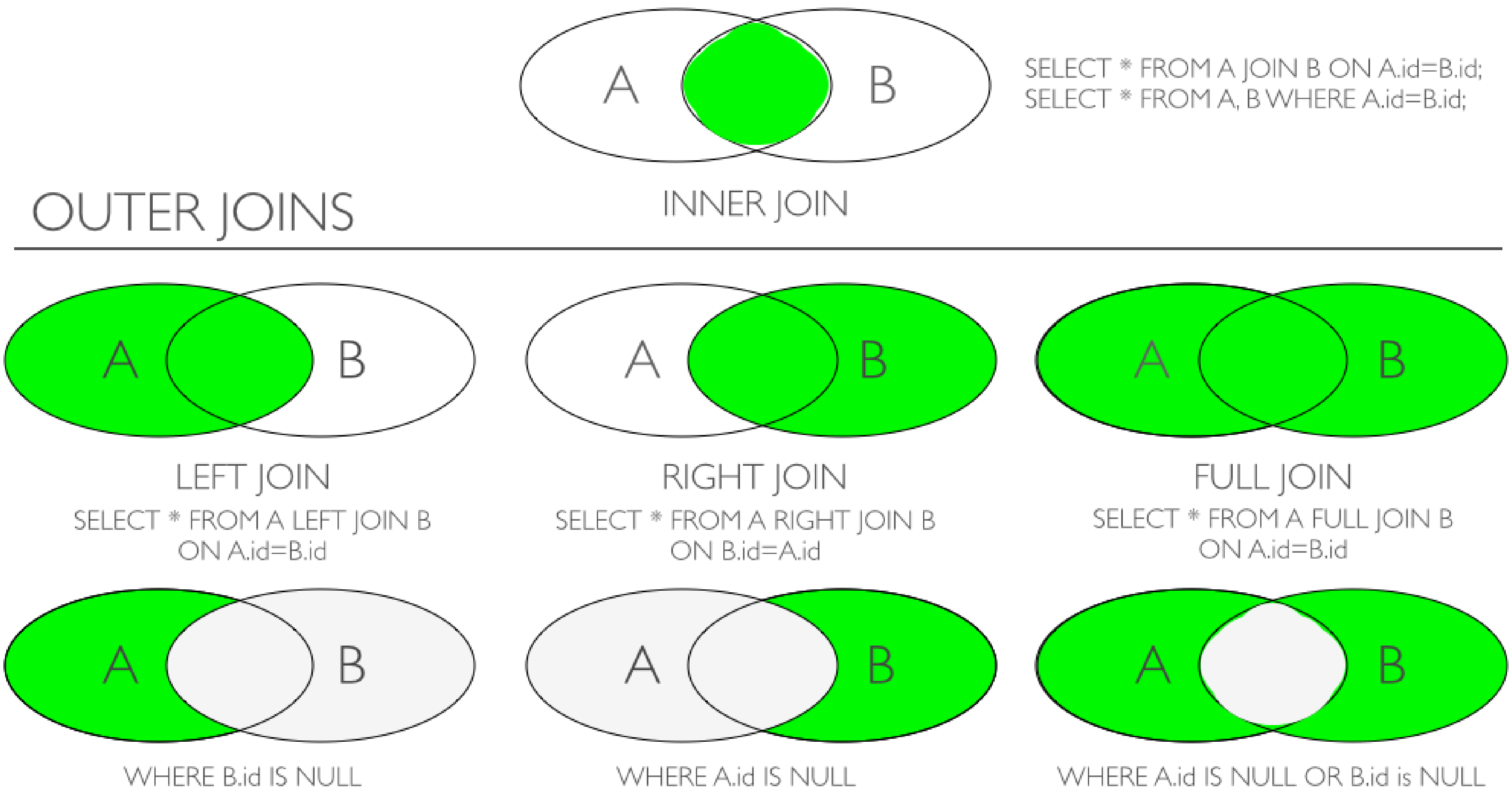
```
SELECT  S.sid
FROM  Sailors S, Reserves R
WHERE  S.sid=R.sid
```

More Joins

- $T = R \bowtie_C S$
  - First build R x S, then apply σ_C
- Generalization of equi-join: A θ B where θ one of =, <, ...
  - Today, more general: σ_C can be any predicate
- Common join types [Quest]:



Even More Joins



"Sailors who've reserved at least 1 boat"

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

- 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 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 string matching
  - `_’ stands for any one character
  - `%’ stands for 0 or more arbitrary characters

"sid's of sailors who have reserved a red or a green boat"

- UNION: Can be used to compute the union of any two union-compatible sets of tuples
  - which themselves are 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' OR B.color='green')
```

- If we replace OR by AND in the first version, what do we get?
- Also available: EXCEPT
  - What do we get if we replace UNION by EXCEPT?

```
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'
```

"Find sid's of sailors who have reserved a red and a green boat"

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

```
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')
```

- Included in the SQL/92 standard, but some systems don't support it

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'
```

- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ!

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 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

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: another set operator, like IN
- If UNIQUE is used, and * is replaced by R.bid:
  finds sailors with at most one reservation for boat #103
  - Why do we have to replace * by R.bid?
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple

More on Set-Comparison Operators

- We have already seen IN, EXISTS and UNIQUE
  - Can also use NOT IN, NOT EXISTS and NOT UNIQUE
- Also available: *op* ANY, *op* ALL, *op* one of $<, >, =, \neq, \leq, \geq$
- "sailors whose rating is greater than that of sailor Horatio"

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

Rewriting INTERSECT Queries Using IN

```
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'
```

- "sailors who've reserved both red & 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
- names of Sailors? replace SELECT S.sid → SELECT S.sname
  - What about INTERSECT query?

Division in SQL

- "sailors who have reserved all boats"
- Let's do it the hard way, without EXCEPT:

(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 ) )
```

(2)

```
                                SELECT  S.sname
                                FROM  Sailors S
Sailors S such that ...         WHERE  NOT EXISTS  (SELECT  B.bid
                                                   FROM  Boats B
  there is no boat B without ...                   WHERE  NOT EXISTS  (SELECT  R.bid
                                                                      FROM  Reserves R
    a Reserves tuple showing S reserved B                             WHERE  R.bid=B.bid
                                                                          AND R.sid=S.sid ) )
```

Aggregate Operators

- Summary information instead of value list

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

A: single column

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

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

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

```
SELECT  AVG ( DISTINCT 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)
```

"Name and age of oldest sailor(s)"

- First query is illegal!
  - We'll look into the reason a bit later, when we discuss GROUP BY
- Sailor age referenced twice in formulation!
- Third query equivalent
  to second query
  - allowed in SQL/92 standard
  - but not supported in some systems

(1)

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

(2)

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

(3)

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

Set Operations: Summary

- SELECT S1.a, S2.b FROM S1, S2
  - $S1 \times S2 = [\langle a,b \rangle \mid a \in S1, b \in S2]$
- S1 UNION S2
  - $S1 \cup S2 = [t \mid t \in S1 \vee t \in S2]$
- S1 INTERSECT S2
  - $S1 \cap S2 = [t \mid t \in S1 \wedge t \in S2]$
- S1 EXCEPT S2
  - $S1 \setminus S2 = [t \mid t \in S1 \wedge t \notin S2]$
- SUM(S.num), AVG(), ...
  - $\sum_{t \in S} t.num$

- EXISTS(S)
  - $S \neq \{\}$
- t IN S2   t = ANY(S2)
  - $t \in S2$
- t *op* ANY(S)   t *op* SOME(S)
  - $\exists\, x \in S: t\ op\ x$
  - $(t\ op\ s_1) \vee ... \vee (t\ op\ s_n)$ for $s_i \in S$
- t *op* ALL (S)
  - $\forall\, x \in S: t\ op\ x$
  - $(t\ op\ s_1) \wedge ... \wedge (t\ op\ s_n)$ for $s_i \in S$

Set Operations: Unique or Duplicates?

- Recall: Relations are multi-sets
- When are duplicates kept / eliminated?

keep duplicates	remove duplicates
SELECT	SELECT DISTINCT
UNION ALL	UNION
INTERSECT ALL	INTERSECT
EXCEPT ALL	EXCEPT

Breaking the Set: ORDER BY

- So far: Query results are (multi) sets, hence unordered
  Sometimes: need result sorted
- ORDER BY clause does this:

```
SELECT      [DISTINCT]  target-list
FROM        relation-list
WHERE       qualification
ORDER BY    sort-list [ASC|DESC]
```

- *sort-list*: list of attributes for ordering (ascending or descending order)
- Ex: "Names of all sailors,
  in alphabetical order"

```
SELECT  S.sname
FROM  Sailors S
ORDER BY  S.sname
```

Grouping

- So far: aggregate operators applied to all (qualifying) tuples.
  Sometimes: apply to each of several groups of tuples
- Consider: "age of the youngest sailor for each rating level"
  - Unknown # of rating levels, and rating values for levels
  - If we knew rating values go from 1 to 10:
    can write loop of 10 queries:

```
For i = 1, 2, ... , 10:
        SELECT  MIN (S.age)
        FROM  Sailors S
        WHERE  S.rating = i
```

…or use GROUP BY:

```
SELECT      MIN( S.age )
FROM        Sailors S
GROUP BY    S.rating
```

Queries With GROUP BY and HAVING

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

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

"Age of the youngest sailor with age ≥ 18, for each rating with at least 2 <u>such</u> sailors"

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

<u>sid</u>	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

"Age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors"

JACOBS UNIVERSITY

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

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

Conceptual Evaluation

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

- compute cross-product of *relation-list*
- discard tuples that fail *qualification*
- delete \`*unnecessary*' attributes
- partition remaining tuples into groups by value of attributes in *grouping-list*
- apply *group-qualification* to eliminate some groups
  - Expressions in *group-qualification* must have a single value per group!
- generate one answer tuple per qualifying group

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

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

rating	minage
7	35.0
8	25.5

What is the result of changing EVERY to ANY?

"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

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

"For each red boat, the number of reservations for this boat"

```
SELECT  B.bid,  COUNT (*) AS scount
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
GROUP BY  B.bid
```

- Grouping over a join of three relations
- What if we remove B.color='red' from the WHERE clause and add a HAVING clause with this condition?

skipping

```
SELECT  B.bid,  COUNT (*) AS scount
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid
 GROUP BY  B.bid
HAVING B.color='red'
```

- What if we drop Sailors and the condition involving S.sid?

```
SELECT  B.bid,  COUNT (*) AS scount
FROM  Boats B, Reserves R
WHERE  R.bid=B.bid AND B.color='red'
GROUP BY  B.bid
```

"Age of the youngest sailor with age ≥ 18, for each rating with at least 2 sailors (of any age)"

```
SELECT  S.rating,  MIN(S.age)
FROM  Sailors S
WHERE  S.age >= 18
GROUP BY  S.rating
HAVING (SELECT  COUNT (*)
           FROM  Sailors S2
           WHERE  S.rating=S2.rating) > 1
```

- Shows HAVING clause can also contain a subquery
- Compare this with the query where we considered only ratings with 2 sailors over 18: What if HAVING clause is replaced by:
  - HAVING COUNT(*) >1

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

"Those ratings for which the average age is the minimum over all ratings"

- Aggregate operations cannot be nested!
  WRONG:

```
SELECT  S.rating
FROM  Sailors S
WHERE  S.age =  (SELECT  MIN (AVG (S2.age))
                 FROM Sailors S2)
```

- Correct solution (in SQL/92):

```
SELECT  Temp.rating, Temp.avgage
FROM  (SELECT  S.rating, AVG (S.age) AS avgage
       FROM  Sailors S
       GROUP BY  S.rating) AS Temp
WHERE  Temp.avgage = (SELECT  MIN (Temp.avgage)
                      FROM  Temp)
```

Null Values

- Field values in a tuple 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
- 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

Integrity Constraints (Review)

- IC describes conditions that every legal instance of a relation must satisfy
  - Inserts/deletes/updates violating ICs disallowed
  - Can be used to ensure application semantics (e.g., *sid* is a key), or prevent inconsistencies (e.g., *sname* has to be a string, *age* must be < 200)
- Types of IC's: Domain constraints, primary key constraints, foreign key constraints, general constraints
  - Domain constraints: Field values must be of right type. Always enforced

General Constraints

- Useful when more general ICs than keys are involved
- Can use queries to express constraint
- Constraints can be named

```
CREATE TABLE   Sailors
(      sid  INTEGER,
       sname  CHAR(10),
       rating  INTEGER,
       age  REAL,
       PRIMARY KEY  (sid),
       CHECK  ( rating >= 1 AND rating <= 10 )
)
```

```
CREATE TABLE  Reserves
(      sname  CHAR(10),
       bid  INTEGER,
       day  DATE,
       PRIMARY KEY  (bid,day),
       CONSTRAINT  noInterlakeRes
       CHECK  (`Interlake' <> ( SELECT  B.bname
                                FROM  Boats B
                                WHERE  B.bid=bid) )
)
```

Assertions

- CHECK constraint is awkward and wrong!
- If Sailors is empty, number of Boats tuples can be *anything*

```
CREATE TABLE   Sailors
(     sid  INTEGER,
      sname  CHAR(10),
      rating  INTEGER,
      age  REAL,
      PRIMARY KEY  (sid),
      CHECK
      (  (SELECT COUNT (S.sid) FROM Sailors S)
      + (SELECT COUNT (B.bid) FROM Boats B) < 100 )
)
```

Number of boats + number of sailors is < 100

- ASSERTION is the right solution: not associated with either table

```
CREATE ASSERTION  smallClub
CHECK
(       (SELECT COUNT (S.sid) FROM Sailors S)
      + (SELECT COUNT (B.bid) FROM Boats B) < 100
)
```

Triggers

JACOBS UNIVERSITY

- Trigger: procedure that starts automatically if & when specified changes occur to the database
- Three parts ("ECA rules"):
  - Event -- activates the trigger
  - Condition -- tests whether the triggers should run
  - Action -- what happens if the trigger runs

Triggers: Example (SQL:1999)

JACOBS UNIVERSITY

```
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 important factor for acceptance of relational model
  - more natural than earlier, procedural query languages
  - Simple, easy-to-grasp paradigm: sets + few generic operations on them
  - Relationally complete = as powerful as relational algebra
    - *in fact, significantly more expressive power than relational algebra*
  - Not *computationally* complete! (no recursion, for example)
- Set orientation good basis for declarative query language
  - Declarative = describe desired result (well, almost :-), more user-oriented
    (imperative = describe algorithm; more implementation-oriented)
- SQL allows specification of integrity constraints
- Triggers respond to changes in the database

Summary (Contd.)

- Many alternative phrasings
  - 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
- NULL for unknown field values brings many complications