SQL

Conceptual Evaluation Strategy

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

- Semantics defined in terms of conceptual evaluation strategy:
  - Compute cross-product of relation-list
  - Discard resulting tuples failing qualification
  - Delete attributes not in target-list
  - If DISTINCT: eliminate duplicate rows
- probably least efficient way to compute query!
  - optimizer will find more efficient strategies

Join

- Join = several tables addressed in one query

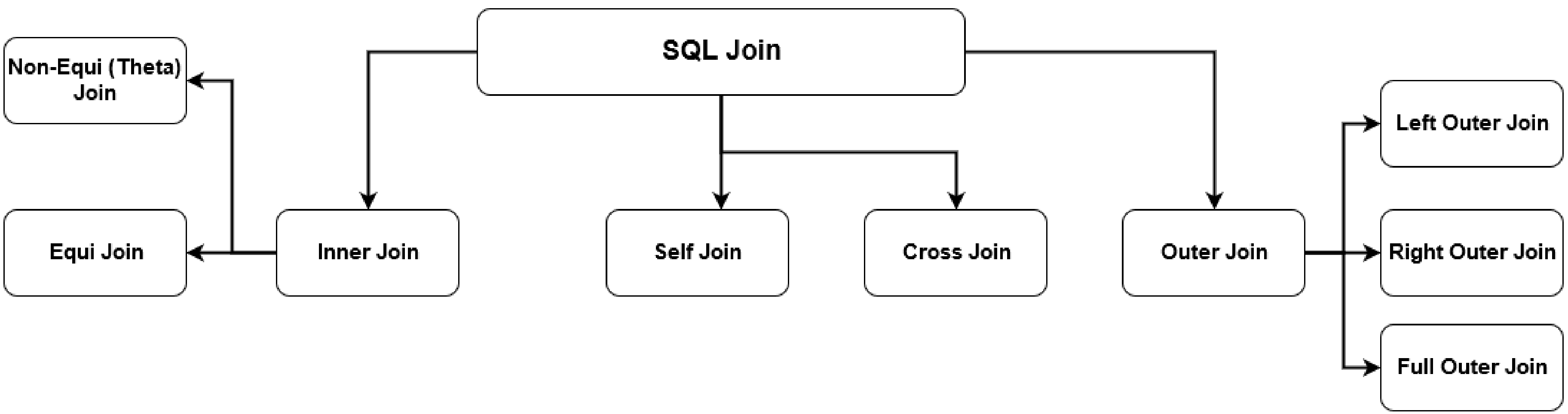
```
SELECT  target-list
FROM  Relation1 R1, Relation2 R2, …
WHERE  qualification
```

- 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:

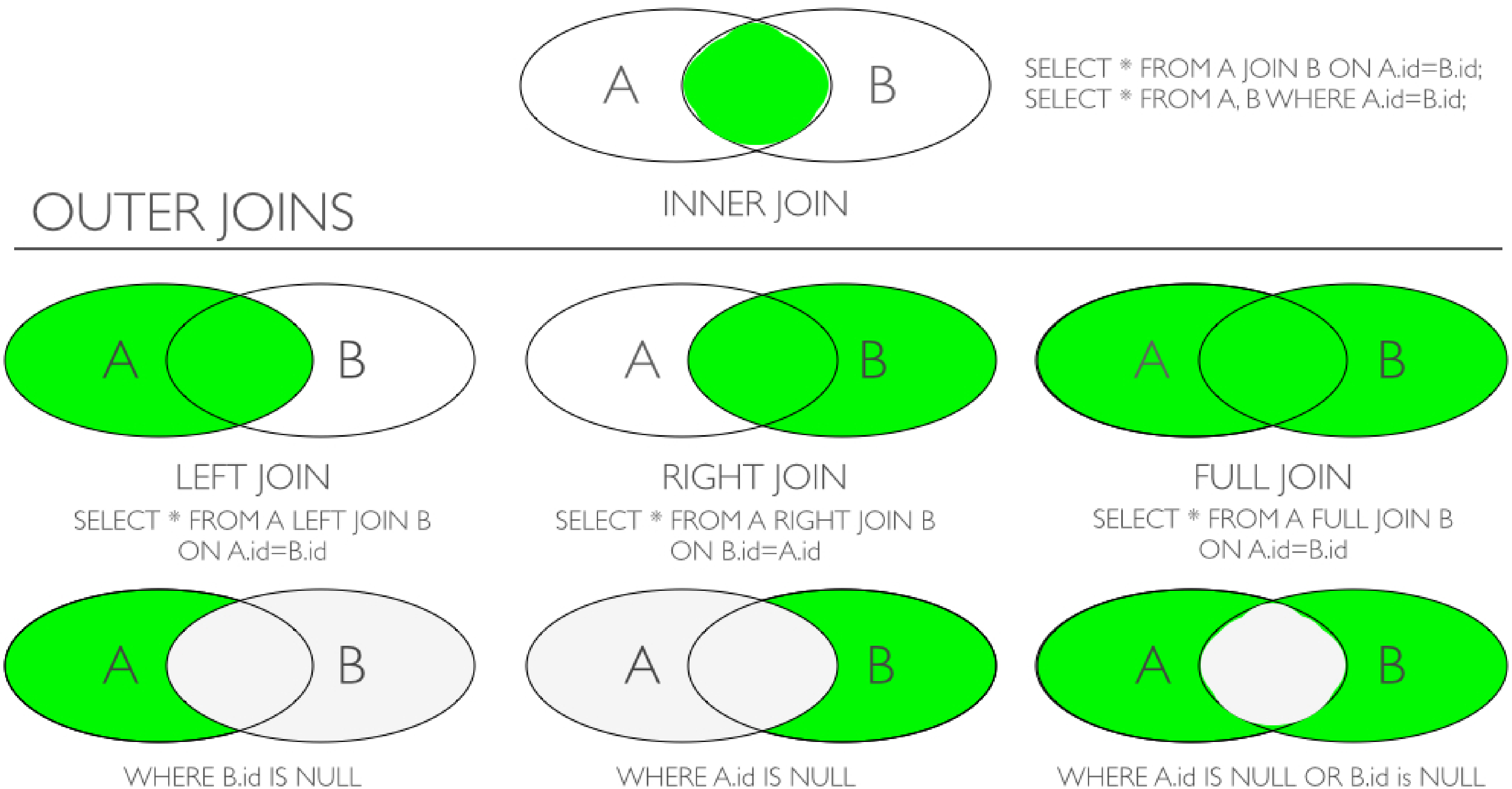
```
SELECT  S.sid
FROM  Sailors S, Reserves R
WHERE  S.sid=R.sid
```

More Joins

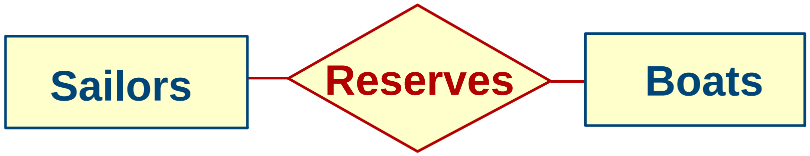
- $T = R \bowtie_C S$
  - First build R x S, then apply σ_C
- Generalization of equi-join: A θ B where θ one of =, <, ...
  - Today, more general: σ_C can be any predicate
- Common join types [Quest]:



Even More on 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

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

(sid)
22
58

- Would adding DISTINCT to this query make a difference?

"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')
```

```
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'
```

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

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

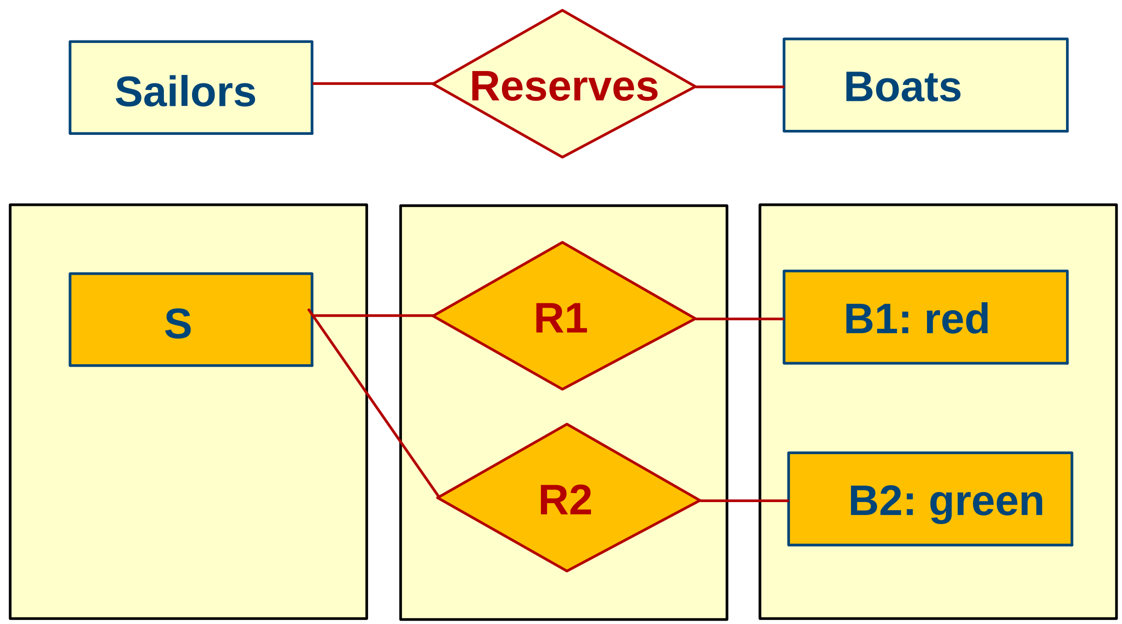
- INTERSECT: Can be used to compute the intersection of any two union-compatible sets of tuples

```
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’)
```

"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')
```



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

CONSTRUCTOR UNIVERSITY

- 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')
```

Key field!

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

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

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)
```

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

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

- "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')
```

Set Difference 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 ) )
```

Constructor University

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

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)
  - $\exists\ x \in S: t\ op\ x$
  - $(t\ op\ s_1) \vee \ldots \vee (t\ op\ s_n)$ for $s_i \in S$
- t *op* ALL (S)
  - $\forall\ x \in S: t\ op\ x$
  - $(t\ op\ s_1) \wedge \ldots \wedge (t\ op\ s_n)$ for $s_i \in S$

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
```

Sailors instance:

<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"

```
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**"

GROUP BY … 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

predicate in WHERE
or in GROUP BY?
Big difference!

"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
```

- HAVING clause can contain subquery
- Compare with 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
```

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

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

- 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: Examples (SQL:1999)

```
CREATE TRIGGER totalMark
AFTER INSERT ON Student
FOR EACH ROW
  INSERT INTO FinalMark VALUES( new.marks )
```

```
CREATE TRIGGER NoSalaryDecrease
BEFORE UPDATE ON Employees
FOR EACH ROW
    BEGIN
        IF NEW.salary < OLD.salary THEN
            ROLLBACK
        END IF;
    END
```

Triggers: Advanced Example

```
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:
    sets + few generic operations on them
  - Relationally complete = as powerful as relational algebra (in fact, more expressive)
  - Not computationally complete!
- Set orientation good basis for declarative query language
  - Declarative vs imperative
- Triggers & constraints
- null

CONSTRUCTOR UNIVERSITY

Key Performance Factors

Mark Fugate • My experience is that proper, or highest normal form normalization takes care of the first half of the optimization process by reducing the size of the stored data and reducing the numbers of operations required to maintain the data.

Query plans and query behaviours tell us how to properly index. Server tuning includes the proper storage media and knowledge of file systems and media tuning. Understanding your servers and knowing how to tune the OS, file systems, storage and kernel is all part of being a DBA.

Further, keeping SQL out of the client code makes all of the above attainable. I force all client applications in our shop to use stored procedures only. This gives me complete control over indexes, table structures, and all queries ensuring that nothing obnoxious enters the database.

1 day ago • Like

- Ref: discussion "what are the key points to improve the query performance" on the LinkedIn Database list

PS: A Moderately Complex Query

```
SELECT stadtbezirk, stadtteil, name, stadtteilchar, 'touche' AS entstehung, the_geom FROM
    (SELECT foo3.stadtbezirk, foo3.stadtteil, foo3.name, foo3.stadtteilchar, foo3.the_geom FROM
        (SELECT foo.gid, max(foo.laengste) AS laengste FROM
            (SELECT a.gid, b.stadtbezirk, b.stadtteil, b.name, b.stadtteilchar,
                (ST_Length(ST_Intersection(a.the_geom, ST_Union(b.the_geom)))) AS laengste
                FROM symdif a, dump b
                GROUP BY a.gid, a.the_geom, b.stadtbezirk, b.stadtteil, b.name, b.stadtteilchar
                HAVING ST_Touches(a.the_geom, ST_Union(b.the_geom))
                ORDER BY a.gid) AS foo
        GROUP BY foo.gid) AS foo2
        ,
        (SELECT a.gid, b.stadtbezirk, b.stadtteil, b.name, b.stadtteilchar, a.the_geom AS the_geom,
            (ST_Length(ST_Intersection(a.the_geom, ST_Union(b.the_geom)))) AS laengste
            FROM symdif a, dump b
            GROUP BY a.gid, a.the_geom, b.stadtbezirk, b.stadtteil, b.name, b.stadtteilchar
            HAVING ST_Touches(a.the_geom, ST_Union(b.the_geom))) AS foo3
    WHERE (foo2.gid = foo3.gid AND foo2.laengste = foo3.laengste)
    GROUP BY foo2.gid, foo3.stadtbezirk, foo3.stadtteil, foo3.name, foo3.stadtteilchar,
        foo3.laengste, foo2.laengste, foo3.the_geom) AS foo4
;
```