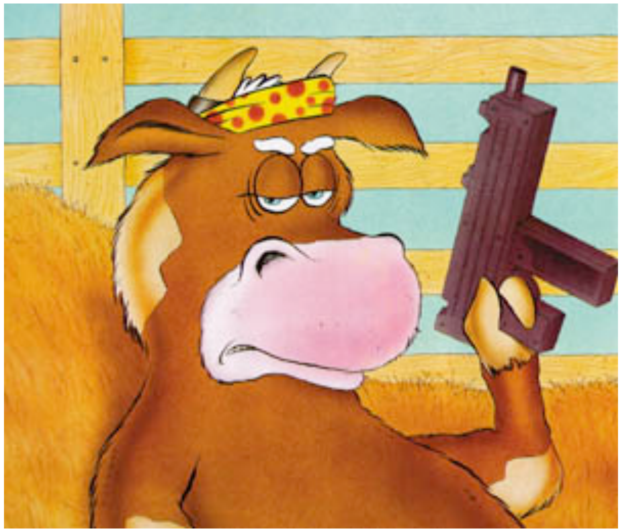
SQL: Queries, Constraints, Triggers
Part 2



Chapter 5.5-5.10

Division in SQL

Find sailors who've reserved all boats.

- 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)
```

All boats

Boats reserved by a given Sailor

(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 tuple showing S reserved B

Division in SQL

Find sailors who've reserved all boats.

For every sailor in sailors do

1. produce a list of every boat
2. from that list, remove the boats that that particular sailor had reserved

If there is nothing in that list (i.e. NOT EXISTS) then the given sailor satisfies the WHERE clause and his or her name can be emitted.

```
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.

```
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))
```

Or,

For every sailor in sailors, emit their name if and only if there are no rows returned when we:

For every boat in boats, emit something and only if there are no rows resulting from:

For every reservation in reserves that matches this particular boat and this particular sailor (from above), emit something.

SQL's Aggregate Operators

- Significant extension of relational algebra.
- Computation and summarization operations
- Appears in *target-list* of query
- Results *aggregate* rather than appear individually

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

single column

- E.x. How many instances in the sailor relation?

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

More examples

- Average age of Sailors with a rating of 10?

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

- Names of all Sailors who have achieved the maximum rating

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

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

More examples (cont)

- How many distinct ratings for Sailors less than 40 years of age?

```
SELECT  COUNT(DISTINCT S.rating)
FROM  Sailors S
WHERE S.age < 40.0
```

- How many reservations were made by Sailors less than 40 years old?

```
SELECT  COUNT(*)
FROM  Sailors S, Reserves R
WHERE  S.sid = R.sid AND S.age < 40
```

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

Find name and age of the oldest sailor(s)

- The first query is invalid! (We’ll look into the reason a bit later, when we discuss GROUP BY.)
- 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, MAX (S.age)
FROM  Sailors S
```

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

```
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 *subgroups*.
- 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
  (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 *qualification* are discarded, *unnecessary* fields are deleted, and the remaining tuples 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*. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.

Basic GROUP BY

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

Answer relation:

```
 rating | minage
--------+--------
      8 |   25.5
      1 |     33
      3 |   25.5
     10 |     16
      9 |     35
      7 |     35
```

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

```
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

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

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

Find age of the youngest sailor with age ≥ 18, for each rating with at least 2 such sailors and where every sailor is 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?

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

Answer relation:

rating	minage
3	25.5
7	35.0
8	25.5

For each red boat, find 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 do we get if we remove *B.color= ‘red’* from the WHERE clause and add a HAVING clause with this condition?
- What if we drop Sailors and the condition involving S.sid?

Find 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  1  <  (SELECT  COUNT (*)
                FROM  Sailors S2
                WHERE  S.rating=S2.rating)
```

- 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

Find those ratings for which the average age is the minimum over all ratings

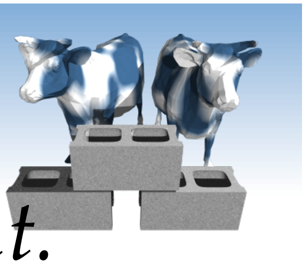
- Aggregate operations cannot be nested! WRONG:

```
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)
```

Find those ratings for which the average age is the minimum over all ratings – cont.



- Some (many?) databases won’t allow correlated subqueries to refer to subqueries in a FROM clause. Solution:

```
SELECT  S.rating, AVG (S.age) AS avgage
INTO temp

FROM  Sailors S

GROUP BY  S.rating ;

SELECT t.rating, t.avgage
FROM temp t
WHERE t.avgage = (SELECT  MIN (t2.avgage) FROM  temp t2) ;
```

Comments

SQL can fall into the trap of being a Write-Only language (i.e., even the query author has trouble reading it later).

```
`temp table of ratings and avg age

SELECT
S.rating, AVG (S.age) AS avgage
INTO temp
FROM  Sailors S
GROUP BY  S.rating ;
```

Sorting

ORDER BY imposes sorting:

```
SELECT  S.rating,  MIN (S.age) AS minage

FROM  Sailors S
WHERE S.age >= 18

GROUP BY  S.rating
ORDER BY S.rating ;
```

gnitroS

ORDER BY DESC sorts in descending order

```
SELECT  S.rating,  MIN (S.age) AS minage

FROM  Sailors S
WHERE S.age >= 18

GROUP BY  S.rating
ORDER BY S.rating DESC;
```

Limiting the rows returned

LIMIT reduces the number of rows returned – very often paired with sorting

```
SELECT  S.rating,  MIN (S.age) AS minage

FROM  Sailors S
WHERE S.age >= 18

GROUP BY  S.rating
ORDER BY S.rating
LIMIT 2;
```

OFFSET – skip some rows first

OFFSET throws away the first few rows

```
' just get the fourth and fifth results
SELECT  S.rating,  MIN (S.age) AS minage

FROM  Sailors S
WHERE S.age >= 18

GROUP BY  S.rating
ORDER BY S.rating
LIMIT 2
OFFSET 3;
```

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.
- 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.

3-valued logic in RDBMS

- =, <>, >, <, >=, <= if either side is null, the result is null
- OR : if either side is true, it's true. If both side are false, it's false. If one side is false and one side is null, it's null.
- AND : the opposite. If both sides are true, it's true. If either side is false, it's false. If both sides are null or if one side is null and the other is true, it evaluates to null.
- DISTINCT treats two nulls as being the same and will coalesce the rows, eliminating the duplicate.
- This means the semantics of IN and EXISTS changes in the presence of nulls.
  - IN is functionally equivalent to = ANY. The test for equality will always fail.
  - EXISTS does a row count of the subquery results, and counts null as a valid row.
  - This makes NOT IN behave non-intuitively as well.

Integrity Constraints (Review)

- An IC describes conditions that every *legal instance* of a relation must satisfy.
  - Inserts/deletes/updates that violate IC's are 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)
```

General Constraints

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

```
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)))
```

Constraints Over Multiple Relations

```
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 plus number of sailors is < 100

- Awkward and wrong!
- If Sailors is empty, the number of Boats tuples can be anything!
- 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 specified changes occur to the DBMS

- Triggers have three parts:
  - *Event* (activates the trigger)
  - *Condition* (tests whether the triggers should run)
  - *Action* (what happens if the trigger runs)

Triggers: Example

- Suppose there was a rule than no one with a rating less than five can reserve a green boat. The following trigger would enforce this rule:

```
CREATE TRIGGER RatingRuleForGreen
   BEFORE INSERT ON Reserves
   BEGIN
       SELECT RAISE(FAIL, 'Sailor is not qualified’ )
       WHERE EXISTS (SELECT * FROM Sailors, Boats
                                   WHERE sid = new.sid AND rating < 5
                                   AND bid = new.bid AND color = 'green');
   END;
```

- Note the special variable “new” for accessing parameters of the original INSERT query

Triggers: Another Example

- Queries of one table can be made to have side-effects in other tables via triggers
- Example “Event Logging”
- We know dates of reservations, but not when they were made. This can be remedied using a trigger as follows:

```
CREATE TRIGGER insertLog AFTER INSERT ON Reserves
BEGIN
    INSERT INTO ReservesLog (sid, bid, resDate, madeDate)
    VALUES (new.sid, new.bid, new.date, DATE( ‘NOW’ );
END;
```

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
- SQL allows specification of rich integrity constraints
- Triggers respond to changes in the database