# *SQL: Queries, Programming, Triggers*

Computer Science Department
Columbia University

# *Example Instances*

***Reserves***

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

***Sailors***

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

***Boats***

| bid | bname | color |
|---|---|---|
| 101 | Interlake | blue |
| 102 | Interlake | red |
| 103 | Clipper | green |
| 104 | Marine | red |

- We will use these instances of the Sailors and Reserves relations in our examples.
- If the key for the Reserves relation contained only the attributes *sid* and *bid,* how would the semantics differ?

# *Basic SQL 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 $<, >, =, \leq, \geq, \neq$ ) 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 *relation-list*.
  - Discard resulting tuples if they fail *qualifications*.
  - Delete attributes that are not in *target-list*.
  - If DISTINCT is specified, eliminate duplicate rows.
- Note: the optimization strategy of a particular DBMS may differ but it must come up with *the same answers*.

# *Example of Conceptual Evaluation*

```
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 | 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  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 most of the time!*

# *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?

# *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
- AS and = are two ways to name fields in result.
- LIKE is used for string matching. `_' stands for any one character and `%' stands for 0 or more arbitrary characters.

## *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).
- EXCEPT/MINUS: set difference.
  - What do we get if we replace UNION by EXCEPT?

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

*Q: what do we get if we replace OR by AND?*

```
SELECT  R.sid
FROM  Boats B, Reserves R
WHERE  R.bid=B.bid
         AND B.color='red'
UNION
SELECT  R.sid
FROM  Boats B, Reserves R
WHERE  R.bid=B.bid
         AND B.color='green'
```

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

- INTERSECT: Can be used to compute the intersection of any two *union-compatible* sets of tuples. (Included in the SQL/92 standard, but some systems don't support it.)
- Contrast symmetry of the UNION and INTERSECT queries with how much the other versions differ.

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

Key field!

```
SELECT  R.sid
FROM  Boats B, Reserves R
WHERE  R.bid=B.bid
       AND B.color='red'
INTERSECT
SELECT  R.sid
FROM  Boats B, Reserves R
WHERE  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; more later.)
- To find sailors who've *not* reserved #103, use NOT IN.
- To understand semantics of nested queries, think of a <u>*nested loops*</u> evaluation: *For each Sailors tuple, check the qualification by computing the subquery.*

# *Example in class*
# *Write SQL using IN*

*Find names of sailors who've 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)
```

How about if we change the query to be:

*Find names of sailors who've not reserved a red boat*

NOT IN

# *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 is another set comparison operator, like IN; it checks for empty set: returns true if table is not empty
  - The example illustrates why, in general, subquery must be re-computed for each Sailors tuple.
- UNIQUE checks for duplicate tuples: returns true if there are no duplicate tuples or it is empty)
  - If UNIQUE is used (instead of EXISTS), and * is replaced by *R.bid*, finds sailors with at most one reservation for boat #103. (* denotes all attributes; Why do we have to replace * by *R.bid*?)

# *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* one of $>, <, =, \geq, \leq, \neq$
- Find sailors whose rating is greater than that of some sailor called Mike:

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

# *Rewriting INTERSECT Queries Using IN*

*Find id/name of sailors who've reserved both a red and a green boat:*

```
SELECT  S.sid, S.sname
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.

# *Division in SQL*

(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))
```

Find sailors who've reserved all boats.

- Let's also 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 tuple 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*

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

```
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)
```

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

# *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.)
- 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
```

# *GROUP BY and HAVING*

- So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.
- 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.

# *Find the 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)
FROM  Sailors S
WHERE  S.age > 18
GROUP BY  S.rating
HAVING  COUNT (*) > 1
```

- Only S.rating and S.age are mentioned in the SELECT, GROUP BY or HAVING clauses; other attributes `*unnecessary*'.
- 2nd column of result is unnamed. (Use AS to name it.)

| sid | sname | rating | age |
|---|---|---|---|
| 22 | dustin | 7 | 45.0 |
| 31 | lubber | 8 | 55.5 |
| 71 | zorba | 10 | 16.0 |
| 64 | horatio | 7 | 35.0 |
| 29 | brutus | 1 | 33.0 |
| 58 | rusty | 10 | 35.0 |

| rating | age |
|---|---|
| 1 | 33.0 |
| 7 | 45.0 |
| 7 | 35.0 |
| 8 | 55.5 |
| 10 | 35.0 |

| rating | |
|---|---|
| 7 | 35.0 |

*Answer relation*

## *Find the 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

# *For each red boat, find the number of reservations for this boat*

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

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

  Only columns that appear in GROUP BY can appear in HAVING, unless they appear as arguments to an aggregate operator in the HAVING clause

# *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 GROUP BY S2.rating)
```

- Correct solution:

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

# *Outer Joins*

- Print the sid of every sailor and the boats the sailor has reserved. Join will not work; sailors with no reservations will not be included.

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

| sid | bid |
|---|---|
| 22 | 101 |
| 58 | 103 |

- Left Outer Join: rows in the left table that do not match some row in the right table appear exactly once, with column from the right table assigned NULL values.

```
SELECT S.sid, R.bid
FROM Sailors S LEFT OUTER JOIN Reserves R ON S.sid = R.sid
```

| sid | bid |
|---|---|
| 22 | 101 |
| 31 | *null* |
| 58 | 103 |

- Right Outer Join: the reverse
- Full Outer Join: both, left and right
- Another example: Print the sid of every sailor and the number of boats the sailor has reserved.

```
SELECT S.sid, COUNT(R.bid)
FROM Sailors S LEFT OUTER JOIN Reserves R ON S.sid = R.sid
GROUP BY S.sid
```

# *Integrity Constraints (Review)*

- Integrity constraints describe conditions that every *legal instance* of a relation must satisfy. It's the DBMS that enforces them on every update – those updates that violates them are disallowed. Types we have seen so far:
  - Domain constraints (field type)
  - Primary key
  - Unique and not null
  - Foreign keys
- We will now discuss general constraints.

# *General Constraints (attribute- & tuple-based)*

- Useful when more general ICs than keys are involved.
- Can use queries to express constraint.
- Constraints can be named.
- Required to hold on nonempty tables only.

```
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)))
```

# *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 )
```

# *Using assertions to enforce total participation*

- All sailors must reserve at least one boat (i.e., total participation of Sailors in Reserves)

```
CREATE ASSERTION  BusySailors
CHECK (
  NOT EXISTS (
    SELECT sid FROM Sailors
    WHERE sid NOT IN (
        SELECT sid FROM Reserves)
  )
)
```

# *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 Issues*

- Before (a change occurs) vs after
- When a statement triggers an event, execution of the trigger's action may occur in one of the following fashion
  - Instead
  - Deferred
  - Asynchronous
- Transactions
  - Part of the statement vs independent action

# *Triggers Example*

```
CREATE TRIGGER init_count BEFORE INSERT ON Students
        DECLARE count INTEGER;
        BEGIN
                count := 0
        END

CREATE TRIGGER incr_count AFTER INSERT ON Students
        WHEN (new.age < 18)  /* also, old (before value) */
        FOR EACH ROW
        BEGIN
                count := count + 1;
        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