SQL: The Query Language

Example Instances

- 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?

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

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.
- 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
```

(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 always!

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?

7

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 and `%' stands for 0 or more arbitrary characters.

8

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).
- 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' 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'
```

9

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 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 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
```

10

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.*

11

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.
- If UNIQUE is used, and * is replaced by *R.bid*, finds sailors with at most one reservation for boat #103. (UNIQUE checks for duplicate tuples; * denotes all attributes. Why do we have to replace * by *R.bid*?)
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple.

12

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* SOME, *op* ALL, *op* IN $>, <, =, \geq, \leq, \neq$
- Find sailors whose rating is greater than that of some sailor called Horatio:

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

13

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

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

14

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
```

15

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
```

16

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*.)

17

Conceptual Evaluation

- The cross-product of *relation-list* is computed, tuples that fail *qualification* are discarded, `*unnecessary*' fields are deleted, as before.
- 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.
- One answer tuple is generated per qualifying group.

18

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

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

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

- 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.)

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

rating	
7	35.0

Answer relation

19

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

```
SELECT B.bid, COUNT (*) AS rcount
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 drop Sailors and the condition involving S.sid?

20

Sorting a Result

- Names, ages of Sailors, sorted by name:
  ```
  SELECT sname, age
  FROM Sailors
  ORDER BY sname
  ```
- Same, ordered by age within a name:
  ```
  SELECT sname, age
  FROM Sailors
  ORDER BY sname, age
  ```
- ORDER BY clause evaluated last

21

Null Values

- Field values in a tuple are sometimes *unknown* (e.g., a rating has not been assigned) or *nonexistent* (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.

22

Summary

- 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 (aggregates, arithmetic, sorting, grouping, string matching….)
- Even queries that can be expressed in RA can often be expressed more naturally in SQL.

23

Summary (cont'd)

- Nulls (*unknown* or *nonexistent*) force a 3-valued logic and odd behavior
- 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.
- SQL3 (SQL:1999) adds nested relational and object-oriented features to SQL (later in course)

24