Lectures 8, 9, 10

SQL: Queries, QBE
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?

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>
Basic SQL Query

• 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
```

<table>
<thead>
<tr>
<th>(sid)</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>(sid)</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
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!
### Example Instances (Large)

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>29</td>
<td>Brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>32</td>
<td>Andy</td>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>58</td>
<td>Rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>64</td>
<td>Horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>71</td>
<td>Zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>74</td>
<td>Horatio</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>Art</td>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/98</td>
</tr>
<tr>
<td>22</td>
<td>102</td>
<td>10/10/98</td>
</tr>
<tr>
<td>22</td>
<td>103</td>
<td>10/8/98</td>
</tr>
<tr>
<td>22</td>
<td>104</td>
<td>10/7/98</td>
</tr>
<tr>
<td>31</td>
<td>102</td>
<td>11/10/98</td>
</tr>
<tr>
<td>31</td>
<td>103</td>
<td>11/6/98</td>
</tr>
<tr>
<td>31</td>
<td>104</td>
<td>11/12/98</td>
</tr>
<tr>
<td>64</td>
<td>101</td>
<td>9/5/98</td>
</tr>
<tr>
<td>64</td>
<td>102</td>
<td>9/8/98</td>
</tr>
<tr>
<td>74</td>
<td>103</td>
<td>9/8/98</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>
Simple SQL Queries

• What is the meaning of and the answer to each of the following queries? (with respect to S2, R2, and B2)

```sql
SELECT S.sname, S.age
FROM Sailors S
WHERE S.rating > 7
```

```sql
SELECT S.sname, S.age
FROM Sailors AS S
WHERE S.rating > 7
```

```sql
SELECT S.sname
FROM Sailors S, Boats B, Reserves R
WHERE S.sid = R.sid AND B.bid = R.bid AND B.color = 'red'
```

```sql
SELECT color
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND sname = 'Lubber'
```
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

```sql
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.
Expressions and Strings (cont.)

- Strings can be compared with the equality operator “==”. The result of comparison ‘Jeff’ = ‘Jeff ‘ is true! (“==” skips over leading and trailing whitespace) However, ‘Jeff ’ LIKE ‘Jeff‘ returns false. Reason: LIKE matches expressions exactly, so need to include a wild-card symbol – ‘Jeff ‘ LIKE ‘Jeff__’ and ‘Jeff ‘ LIKE ‘Jeff%’ both return true.

- SQL 1999 includes an even more powerful operator SIMILAR which allows regular expressions as patterns for searching text.

SELECT S1.sname AS name1, S2.sname AS name2 FROM Sailors S1, Sailors S2 WHERE 2*S1.rating = S2.rating-1

SELECT S.age FROM Sailors S WHERE S.sname LIKE ‘%Peter%’
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, analogous to the difference operator (What do we get if we replace UNION by EXCEPT?)

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
  AND (B.color='red' OR B.color='green')

SELECT S.sid
FROM Sailors S, Boats B, Reserves R
  AND B.color='red'
UNION
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
  AND B.color='green'
```
Find sid’s of sailors who’ve reserved a red \textit{and} a green boat

- **INTERSECT**: Can be used to compute the intersection of any two \textit{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.
- What issue arises if we replace \textit{S.sid} by \textit{S.sname} in the second query?

```sql
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' \textbf{AND} B2.color='green')
```

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND B.color='red'
INTERSECT
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
AND B.color='green'
```
Find names of sailors who’ve reserved a red \textit{and} a green boat

- If we change $R.\text{sid}$ by $S.\text{sname}$, a subtle problem emerges if two different sailors with the same name reserved one a red and the other a green boat. The solution – use a nested query!

- The Sailors relation isn’t really necessary for the $sids$ version of the query

\begin{verbatim}
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'
\end{verbatim}
Find all sids of sailors who have a rating of 10 or reserved boat 104

```
SELECT S.sid
FROM Sailors S
WHERE S.rating = 10
UNION
SELECT R.sid
FROM Reserves R
WHERE R.bid=104
```

- **UNION**, **INTERSECT**, and **EXCEPT** do not preserve duplicates! Use **UNION ALL**, **INTERSECT ALL**, and **EXCEPT ALL** to keep duplicate tuples in the answer.
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.*
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. EXISTS checks for set non-emptiness, while IN is like a set membership operator.
- 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 \(R.bid\)?) See next slide.
- Illustrates why, in general, subquery must be re-computed for each Sailors tuple.
Find all sailors who have reserved boat #103 at least twice

- A long solution, requiring a three-way join:

  ```sql
  SELECT S.sname
  FROM Sailors S, Reserves R1, Reserves R2
  WHERE (S.sid = R1.sid) AND (S.sid = R2.sid) AND
        (R1.bid = R2.bid) AND (R1.day <> R2.day) AND
        (R1.bid = 103)
  ```

- A solution using a nested query with correlation:

  ```sql
  SELECT S.sname
  FROM Sailors S
  WHERE NOT UNIQUE (SELECT R.bid
                     FROM Reserves R
                     WHERE R.bid=103 AND S.sid=R.sid)
  ```
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 $>, <, =, \ge, \le, \ne$
- IN $\equiv$ ANY, NOT IN $\equiv$ <> ALL
- Find sailors whose rating is greater than that of some sailor called Horatio:

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

Find the names of sailors who have not reserved a red boat:

SELECT S.sname
FROM   Sailors S
WHERE  S.sid NOT IN ( SELECT R.sid
                      FROM   Reserves R
                      WHERE  R.bid IN ( SELECT B.bid
                                      FROM   Boats B
                                      WHERE B.color = 'red')
                      )

• Include a sailor in the answer if his/her sid does not appear in a reservation of a red boat.
Rewriting INTERSECT Queries Using IN

Find 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’)

• Similarly, EXCEPT queries re-written using NOT IN.
• To find names (not sid’s) of Sailors who’ve reserved both red and green boats, just replace S.sid by S.sname in SELECT clause. (What about INTERSECT query?)
Division in SQL

Find sailors who’ve 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
    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 ... WHERE NOT EXISTS (SELECT R.bid
                                          FROM Reserves R
                                          WHERE R.bid=B.bid
                                          AND R.sid=S.sid))

there is no boat B without ...

a Reserves tuple showing S reserved B
Find the sailors with the highest rating

• Can express with a nested query and an ALL set operator.

```
SELECT S.sid
FROM Sailors S
WHERE S.rating >= ALL ( SELECT S2.rating
                        FROM Sailors S2 )
```

• What happens if we replace ALL with ANY? How do we ask for the sailors with the lowest rating?

• SQL adds much more power and flexibility with its aggregate operations, which are not available in relational algebra.
Aggregate Operators

- Significant extension of relational algebra.

SELECT COUNT (*)
FROM Sailors S
SELECT S.sname
FROM Sailors S
SELECT AVG (S.age) WHERE S.rating= (SELECT MAX(S2.rating)
FROM Sailors S2)
SELECT COUNT (DISTINCT S.sname)
FROM Sailors S
WHERE S.sname=’Bob’

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

single column
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.

```sql
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 $i = 1, 2, \ldots, 10$:
  
  ```sql
  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 1999 introduces two new operators EVERY and ANY that, while still specify conditions to be satisfied by each group, test individual tuples in the group rather than compute aggregate functions.

• 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 such 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.)

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

Answer relation
Alternative Formulations

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

• Or we can use a nested subquery in the FROM clause. Note that we must give the result a name with the AS keyword.

SELECT Temp.rating, Temp.minage
FROM ( SELECT S.rating,
        AVG (S.age) AS minage,
        COUNT(*) AS ratingcount
    FROM Sailors S
    WHERE S.age >= 18
    GROUP BY S.rating ) AS Temp
WHERE Temp.ratingcount > 1
For each red boat, find the number of reservations for this boat

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

• What do we get if we remove $B.color = 'red'$ from the WHERE clause and add a HAVING clause with this condition? (Answer: Illegal! Why?)
Find all sailors who have reserved boat #103 at least twice (revisited)

• A solution (#3) using GROUP BY and HAVING:

```sql
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                FROM Reserves R
                WHERE R.bid = 103
                GROUP BY R.sid
                HAVING COUNT(*) > 1)
```

• An uncorrelated efficient query!
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
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)
```
Ordering the results

• The above query lists all sailors named ‘Horatio’ in the order of smallest to largest rating:
  ```sql
  SELECT S.sid, S.rating
  FROM Sailors S
  WHERE S.sname = 'Horatio'
  ORDER BY S.rating
  ```

• Default order is ascending (alphabetical). Can specify desired order:
  – ASC = ascending, DESC = descending

• For duplicate values, may wish to specify a secondary sort value:
  ```sql
  SELECT S.sid, S.sname
  FROM Sailors S
  WHERE S.rating = 10
  ORDER BY S.sname ASC, S.sid DESC
  ```
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., how should WHERE clause qualification handle rows for which the comparison returns unknown?)
  – New operators (in particular, *outer joins*) possible/needed.
Null Values in SQL

- Arithmetic comparisons involving null values always return *unknown* (e.g., a select on Sailors for which rating > 8 will not return those sailor tuples whose rating is null).
- SQL provides the operator IS NULL to test whether a field is null (IS NOT NULL is defined as well).
- Nulls in boolean expressions:
  - OR involving only one null value operand evaluates to the boolean value of the other operand, two null values always returns *unknown*.
  - AND evaluates to unknown if at least one operand is null.
  - NOT evaluates to unknown for null-valued operands.
- WHERE clause qualification eliminates all rows for which the qualification criterion does not evaluate to true.
Outer Joins

- Do not require that each record in the two joined tables has a matching record in the other table - the record from one relation is retained in the joined table if no matching record in the other relation exists.
- Variations: left outer join, right outer join, full outer join.

```sql
SELECT  S.sid, R.bid
FROM    Sailors S  NATURAL LEFT OUTER JOIN  Reserves R
SELECT *
FROM    Manager
        RIGHT OUTER JOIN  Department
        ON    Manager.did = Department.did
```
Integrity Constraints

- 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.

```
CREATE DOMAIN ratingval INTEGER DEFAULT 1
CHECK ( VALUE >= 1 AND VALUE <= 10 )
```
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)))
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 )

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

Number of boats plus number of sailors is < 100
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)
    - Prevent or undo the event
    - Perform some sequence of DB operations
Triggers (cont.)

• Can specify whether trigger action should be executed before or after the activating event (other options also available, such as instead)
• Can define trigger action to execute once per activating statement or once per modified record (with FOR EACH ROW – row-level trigger, and FOR EACH STATEMENT – statement-level trigger, clauses)
• Keywords OLD and NEW can be used to refer to the values before and after modification
• SQL 1999 adds set-oriented triggers by allowing to reference sets of new tuples with NEW TABLE keyword
Triggers (cont.)

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)
FOR EACH ROW
BEGIN
    count := count + 1;
END
Triggers (cont.)

CREATE TRIGGER netWorthTrigger
AFTER UPDATE OF netWorth ON MovieExec
REFERENCING
  OLD AS OldTuple
  NEW AS NewTuple
WHEN (OldTuple.netWorth > NewTuple.netWorth)
FOR EACH ROW
  UPDATE MovieExec
    SET netWorth = OldTuple.netWorth
    WHERE exec_id = NewTuple.exec_id;

• Prevents any changes to the ‘netWorth’ attribute of MovieExec relation!
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
Triggers: Issues

• Chain activations – Recursive triggers
• Triggers vs. Integrity Constraints
  – Constraints – easier to understand, optimize execution; execute upon any change to the data vs. a specified trigger event
  – Triggers – more flexible ways to ensure integrity
    • Orders(itemid, quantity, customerid, unitprice) – trigger can fetch unitprice from the Items table and auto fill-in the correct data
    • Check for customer purchases exceeding the current credit limit, define flexible policies to handle these situations
    • Can use for generating logs – auditing, security, statistics
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 (cont.)

- NULL for unknown field values brings many complications
- SQL allows specification of rich integrity constraints
  - Tuple-based CHECK constraints
  - Global ASSERTION constraints
- Triggers respond to changes in the database
QBE: Query-By-Example

• Another relationally complete query language
  – GUI in nature: users write queries by creating example tables on the screen
  – Convenient for queries that involve only a few tables and are not too complex
  – Developed at IBM but QBE interfaces are now available in many DBMS’s, such as Paradox and MS Access; often offered in addition to SQL
QBE (cont.)

- QBE uses variables with domains determined by the columns in which they appear.
- Variable names are prefixed with ‘_’.
- Fields to appear in the answer are specified by the command P. (print).
- Variable names that are used only once can be omitted.

<table>
<thead>
<tr>
<th>Sailors</th>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td></td>
<td></td>
<td>P._N</td>
<td>P._A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sailors</th>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>P.</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
QBE (cont.)

- To eliminate duplicates, put UNQ. under the relation name.
- To sort use .AO and .DO commands in conjunction with P. An optional argument can be used to sort on more than one field.
- To join two tables put the same variable name in the columns of both tables.

<table>
<thead>
<tr>
<th>Sailors</th>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P.</td>
<td></td>
<td>P.AO(2)</td>
<td>P.AO(1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sailors</th>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_Id</td>
<td>P._S</td>
<td></td>
<td>&gt;25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reserves</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_Id</td>
<td></td>
<td>9/18/98</td>
</tr>
</tbody>
</table>
QBE (cont.)

• Find the colors of all ‘Interlake’ boats reserved by sailors aged > 25 on 8/24/96

<table>
<thead>
<tr>
<th>Sailors</th>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_Id</td>
<td></td>
<td></td>
<td>&gt;25</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reserves</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_Id</td>
<td>_B</td>
<td>8/24/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Boats</th>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>_B</td>
<td>Interlake</td>
<td>P.</td>
<td></td>
</tr>
</tbody>
</table>

• Print the names and ages of sailors who have reserved some boat that is also reserved by the sailor with id 22

<table>
<thead>
<tr>
<th>Sailors</th>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_Id</td>
<td>P._N</td>
<td>P._A</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reserves</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_Id</td>
<td>_B</td>
<td>22</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>bid</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>_B</td>
<td></td>
</tr>
</tbody>
</table>