Recap: Basic SQL Query

**relation-list**: a list of relations

**target-list**: a list of attributes of tables in **relation-list**

**qualification**: comparisons using AND, OR and NOT

comparisons are: `<attr> <op> <const> or <attr1> <op> <attr2>`, where **op** is:

\[
<, >, \leq, \geq, \neq
\]

**DISTINCT**: optional, removes duplicates

By default SQL SELECT does **not** eliminate duplicates! (“multiset”)
Recap: Query Semantics

Conceptually, a SQL query can be computed:

1. **FROM**: compute cross-product of tables (e.g., Students and Enrolled)
2. **WHERE**: Check conditions, discard tuples that fail (applying “selection” condition)
3. **SELECT**: Delete unwanted fields (applying “projection”)
4. If **DISTINCT** specified, eliminate duplicate rows

probably the least efficient way to compute a query!

Query Optimization finds the *same answer* more efficiently
Recap: Range Variables

```
SELECT sname
FROM Sailors, Reserves
WHERE Sailors.sid = Reserves.sid AND bid = 103
```

can be rewritten using range variables as:

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND bid = 103
```

Can use Range Variables – do not need though. Why?
Recap: Expressions

Use **AS** to provide column names

```
SELECT S.age, S.age-5 AS age1, 2*S.age AS age2
FROM    Sailors S
WHERE   S.sname = 'dustin'
```

Can also have **expressions** in WHERE clause:

```
SELECT S1.sname AS name1, S2.sname AS name2
FROM    Sailors S1, Sailors S2
WHERE   2*S1.rating = S2.rating - 1
```
Recap: String operations

SQL also supports some string operations

“LIKE” is used for string matching.

```
SELECT S.age, age1=S.age-5, 2*S.age AS age2
FROM Sailors S
WHERE S.sname LIKE 'B_%B'
```

’_’ stands for any one character

’%’ stands for 0 or more arbitrary characters

>, < string comparison is supported by most systems
Recap: Nested Queries

WHERE clause can itself contain an SQL query!

```
SELECT S.sname
FROM  Sailors S
WHERE  S.sid IN (SELECT R.sid
                 FROM    Reserves R
                 WHERE   R.bid=103)
```
Recap: Nested Queries with Correlation

Subquery must be recomputed for each Sailors tuple.

Think of subquery as a function call that runs a query!

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS (SELECT *
               FROM Reserves R
               WHERE R.bid=103 AND S.sid=R.sid)
```
Recap: Set Operations

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'

INTERSECT

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 R.bid=B.bid
    AND B.color='green'
Recap: ANY and ALL Set-Comparison Operators

Find sailors with rating greater than the rating of at least one sailor called ‘Horatio’:

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

Find sailors with rating greater than the rating of all 20-year old sailors:

```sql
SELECT  *
FROM    Sailors S
WHERE   S.rating > ALL (SELECT  S2.rating
                         FROM    Sailors S2
                         WHERE S2.age = 20)
```
Recap: Set-Difference using NOT IN

Find all sailors who have not reserved a red boat

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

Nested – NO correlation!
Recap: Set-Difference using NOT EXISTS

Find all sailors who have **not** reserved a red boat

```sql
SELECT S.sid
FROM Sailors S
WHERE NOT EXISTS
  (SELECT *
   FROM Reserves R, Boats B
   WHERE R.sid = S.sid
     AND R.bid = B.bid
     AND B.color = 'red')
```

Nested – correlation!
Aggregate Operators

Significant extension of relational algebra.

```
SELECT COUNT (*)
FROM Sailors S

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

SELECT COUNT (DISTINCT S.rating)
FROM Sailors S
WHERE S.sname='Bob'
```
Find name and age of the oldest sailor(s)

The first query is incorrect!

Third query equivalent to second query allowed in SQL/92 standard, but not supported in some systems.
ARGMAX?

The Sailor with the highest rating
What about ties for highest?

\[
\begin{align*}
&\text{SELECT } * \\
&\text{FROM } \text{Sailors } S \\
&\text{WHERE } S.\text{rating} \geq \text{ALL} \\
&(\text{SELECT } S2.\text{rating} \\
&\text{FROM } \text{Sailors } S2)\\
&\text{SELECT } * \\
&\text{FROM } \text{Sailors } S \\
&\text{WHERE } S.\text{rating} = \\
&(\text{SELECT } \text{MAX}(S2.\text{rating}) \\
&\text{FROM } \text{Sailors } S2)\\
&\text{SELECT } * \\
&\text{FROM } \text{Sailors } S \\
&\text{ORDER BY } \text{rating DESC} \\
&\text{LIMIT } 1;
\end{align*}
\]
Division in SQL

Find sailors who have reserved all boats.

Sailors $S$ for which ...

```
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)
                 )
```

$a$ Reserves tuple $AND$ $R$.sid=$S$.sid$)

showing $S$ reserved $B$
SQL DDL
Recap: SQL DDL

CREATE TABLE Enrolled
    (sid CHAR(20),
     cid CHAR(20),
     grade CHAR(2),
     PRIMARY KEY (sid,cid),
     FOREIGN KEY (sid) REFERENCES Students )
SQL DDL – General Constraints

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
  PRIMARY KEY (sid,cid),
  FOREIGN KEY (sid) REFERENCES Students,
  CHECK grade LIKE 'A' OR grade LIKE 'B'
   OR grade LIKE 'C' OR grade LIKE 'D')
SQL DDL – General Constraints

CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid,cid),
FOREIGN KEY (sid) REFERENCES Students,
CONSTRAINT checkGrade
CHECK (grade LIKE 'A' OR grade LIKE 'B'
    OR grade LIKE 'C' OR grade LIKE 'D') )
SQL DDL – General Constraints

CREATE TABLE Enrolled
    (sid CHAR(20),
     cid CHAR(20),
     grade CHAR(2),
    PRIMARY KEY (sid,cid),
    FOREIGN KEY (sid) REFERENCES Students,
    CONSTRAINT checkNumber
    CHECK ((SELECT COUNT (sid) FROM Students) +
            (SELECT COUNT DISTINCT (cid) FROM Enrolled) < 1000 ) )
JOINS
Joins

```
SELECT (column_list)
FROM   table_name
       [INNER  | NATURAL  | {LEFT  | RIGHT  | FULL}  | {OUTER}]
JOIN   table_name
    ON qualification_list
WHERE  ...
```

*INNER is default*

```
SELECT sname FROM sailors S JOIN reserves R ON S.sid=R.sid;
```

```
SELECT sname FROM sailors S NATURAL JOIN reserves R
WHERE R.bid = 102;
```
Inner Joins

\[
\text{SELECT } s\text{.sid, } s\text{.sname, } r\text{.bid} \\
\text{FROM Sailors } s, \text{ Reserves } r \\
\text{WHERE } s\text{.sid} = r\text{.sid}
\]

\[
\text{SELECT } s\text{.sid, } s\text{.sname, } r\text{.bid} \\
\text{FROM Sailors } s \text{ INNER JOIN Reserves } r \\
\text{ON } s\text{.sid} = r\text{.sid}
\]

They are equivalent!
**Left Outer Join**

Returns all matched rows, plus all unmatched rows from the table on the **left** of the join clause

(use nulls in fields of non-matching tuples)

```sql
SELECT s.sid, s.sname, r.bid
FROM Sailors s LEFT OUTER JOIN Reserves r
ON s.sid = r.sid;
```

Returns all sailors & bid for boat in any of their reservations

Note: no match for s.sid? r.sid IS NULL!
### SQL Query

```sql
SELECT s.sid, s.sname, r.bid
FROM Sailors s LEFT OUTER JOIN Reserves r
ON s.sid = r.sid;
```

### Table 1: Sailors

<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>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</td>
</tr>
</tbody>
</table>

### Table 2: Reserves

<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>95</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

### Table 3: Result

<table>
<thead>
<tr>
<th>s.sid</th>
<th>s.name</th>
<th>r.bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Dustin</td>
<td>101</td>
</tr>
<tr>
<td>95</td>
<td>Bob</td>
<td>103</td>
</tr>
<tr>
<td>31</td>
<td>Lubber</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Right Outer Join

Returns all matched rows, plus all unmatched rows from the table on the **right** of the join clause

(use nulls in fields of non-matching tuples)

```
SELECT r.sid, b.bid, b.bname
FROM Reserves r RIGHT OUTER JOIN Boats b
ON r.bid = b.bid;
```

Returns all boats & information on which ones are reserved

Note: no match for b.bid? r.bid IS NULL!
Full Outer Join

Full Outer Join returns all (matched or unmatched) rows from the tables on both sides of the join clause

```
SELECT r.sid, b.bid, b.bname
FROM Reserves2 r FULL OUTER JOIN Boats2 b
  ON r.bid = b.bid;
```

Returns all boats & all information on reservations

No match for r.bid?
  - b.bid IS NULL AND b.bname is NULL

No match for b.bid?
  - r.sid is NULL
GROUP BY AND HAVING
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 (!):

\[
\text{SELECT MIN (S.age)} \\
\text{FROM Sailors S} \\
\text{WHERE S.rating = i}
\]

For \( i = 1, 2, \ldots, 10 \):
Queries With GROUP BY and HAVING

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
[HAVING group-qualification]
```

Group rows by columns in `grouping-list`
Every column from `target-list` mast appear in the `grouping-list`
HAVING restricts through an aggregate which group-rows are part of the result
Conceptual Evaluation

1. Cross-product of relation-list
2. Select only tuples that follow the where clause qualification
3. Partition rows by the value of attributes in grouping-list
4. Select only groups that follow the group-qualification

Attributes in target-list must also be in grouping-list.

Expressions in group-qualification must have a single value per group! That is, attributes in group-qualification must be part of an aggregate op / must appear in the grouping-list.
Find the age of the youngest sailor with age \( \geq 18 \), 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
```
Can you do this using Group By and Having?

```
SELECT S.name
FROM Sailors S, Reserves R
WHERE S.sid = R.sid
GROUP BY S.name, S.sid
HAVING COUNT(DISTINCT R.bid) =
(Select COUNT (*) FROM Boats)
```

Note: must have both sid and name in the GROUP BY clause. Why?

1. Attributes in target-list must also be in grouping-list.
2. Expressions in group-qualification must have a single value per group!
SELECT S.name, S.sid
FROM Sailors S, reserves R
WHERE S.sid = R.sid
GROUP BY S.name, S.sid
HAVING COUNT(DISTINCT R.bid) =
(Select COUNT(*) FROM Boats)

s.name   s.sid   r.sid   r.bid
Dustin    22    22    101
Lubber    31    22    101
Bob       95    22    101
Dustin    22    95    102
Lubber    31    95    102
Bob       95    95    102

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

Count (*) from boats = 4

Apply having clause to groups

s.name   s.sid
Dustin    22
Bob       95

s.name   s.sid
Bob       95
Sorting the Results of a Query

ORDER BY column [ ASC | DESC ] [, ...]

```sql
SELECT S.rating, S.sname, S.age
FROM Sailors S, Boats B, Reserves R
AND B.color='red'
ORDER BY S.rating, S.sname;
```

Extra reporting power obtained by combining with aggregation.

```sql
SELECT S.sid, COUNT(*) AS redrescnt
FROM Sailors S, Boats B, Reserves R
AND B.color='red'
GROUP BY S.sid
ORDER BY redrescnt DESC;
```
Summary: The SQL Query

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
ORDER BY attribute-list
```