Structured Query Language (SQL)

Yanlei Diao
UMass Amherst

Slides Courtesy of R. Ramakrishnan, J. Gehrke, and G. Miklau
Structured Query Language (SQL)

- **Foundation**
  - Semantics is based on relational calculus
  - Evaluation is based on relational algebra
  - Data model is a multiset model (extension of a set model)

1. **Data Manipulation Language (DML)**
   - posing queries
   - operating on tuples

2. **Data Definition Language (DDL)**
   - operating on tables/views
SQL Overview

- Query capabilities
  - SELECT-FROM-WHERE blocks
  - Set operations (union, intersect, except)
  - Nested queries (correlation)
  - Ordering
  - Aggregation and grouping
  - Null values

- Database updates
- Tables and views
- Integrity constraints
### Example Instances

<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>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</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>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</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 each with a range-variable.
- **qualification**: predicates combined using AND, OR and NOT.
  - predicate: \( \text{attr} \ op \ \text{const} \) or \( \text{attr1} \ op \ \text{attr2} \), \( op: <, >, \geq, \leq, =, <> \)
- **target-list**: a list of attributes of relations in **relation-list**
  - **DISTINCT** indicates no duplicates in the answer. Default is that duplicates are \textit{not} eliminated!
  - SQL uses a **multiset**-based model!
Conceptual Evaluation Strategy

<table>
<thead>
<tr>
<th>SELECT</th>
<th>[DISTINCT] target-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>relation-list</td>
</tr>
<tr>
<td>WHERE</td>
<td>qualification;</td>
</tr>
</tbody>
</table>

- **relation-list**: cross-product (×)
- **qualification**: selection (σ)
- **target-list**: projection (π)
  - duplicate elimination if DISTINCT

- This is possibly the least efficient way to execute the query! Leave the issue to Query Optimizer…
An Example SQL Query

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

What is the relational algebra for this query?

\[ \pi_{sname}((\sigma_{bid=103} \text{Reserves}) \bowtie \text{Sailors}) \]
```sql
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND R.bid=103;
```
A Note on Range Variables

- Really needed only if the same relation appears twice in the FROM clause.

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

OR

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

It is good style, however, to use range variables always!
Q1: What Does the Query Compute?

SELECT  S.sid
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid;

Q2: Would adding **DISTINCT** to this query make a difference?

Q3: What if we replace *S.sid* by *S.sname* in the SELECT clause and then add **DISTINCT**? Compare num of results with Q2.

SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid;
String Pattern Matching

```
SELECT  S.age
FROM    Sailors S
WHERE   S.sname  LIKE  'A_%M';
```

- Find the ages of sailors whose names begin with ‘A’, end with ‘M’, and contain at least one character between ‘A’ and ‘M’.

- LIKE is used for string matching.
  - ‘_’ stands for any single character.
  - ‘%’ stands for 0 or more arbitrary characters.
Arithmetic Expressions

SELECT  S.age, age1 = S.age-5, 2*S.age AS age2
FROM    Sailors S
WHERE   S.sname LIKE ‘A%M’;

- Find triples (of ages of sailors and two fields defined by expressions) for sailors whose names begin with ‘A’ and end with ‘M’.
- AS and = are two ways to name fields in the result.
- Arithmetic expressions can also appear in the predicates in WHERE.
SQL Overview

- Query capabilities
  - SELECT-FROM-WHERE blocks
  - Set operations (union, intersect, except)
  - Nested queries (correlation)
  - Ordering
  - Aggregation and grouping
  - Null values
- Database updates
- Tables and views
- Integrity constraints
Find sid’s of sailors who’ve reserved a red or a green boat

- If we replace OR by AND in this query, what do we get?

SELECT DISTINCT R.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
    AND (B.color=‘red’ OR B.color=‘green’).

- UNION: computes the union of any two union-compatible sets of tuples (which are themselves the result of SQL queries).
  - Duplicates after UNION?
  - What if we remove the DISTINCT keyword?

SELECT DISTINCT R.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
    AND B.color=‘red’
UNION
SELECT DISTINCT R.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
    AND B.color=‘green’;
Find sid’s of sailors who’ve reserved a red and a green boat

- **INTERSECT**: computes the intersection of any two *union-compatible* sets of tuples.
  - Duplicates after INTERSECT?
  - What if we remove the **DISTINCT** keyword?

```sql
SELECT DISTINCT R.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
     AND B.color=‘red’
INTERSECT
SELECT DISTINCT R.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
     AND B.color=‘green’;
```
Find sid’s of sailors who’ve reserved a red and a green boat

- INTERSECT is only a derived operator, we can rewrite it:

```
SELECT DISTINCT R1.sid
FROM Reserves R1, Boats B1,
    Reserves R2, Boats B2
WHERE R1.bid=B1.bid
    AND B1.color=‘red’
INTERSECT
SELECT DISTINCT R.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
    AND B.color=‘green’;
```

Need DISTINCT to be equivalent!

```
SELECT DISTINCT R1.sid
FROM Reserves R1, Boats B1,
    Reserves R2, Boats B2
WHERE R1.bid=B1.bid AND B1.color=‘red’
    AND R1.sid=R2.sid;
```
Find sid’s of sailors who’ve reserved …

- Also available: **EXCEPT** (What does this query return?)

```sql
SELECT DISTINCT S.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
AND B.color=‘green’
EXCEPT
SELECT DISTINCT S.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
AND B.color=‘red’;
```
SQL Overview

- Query capabilities
  - SELECT-FROM-WHERE blocks
  - Set operations (union, intersect, except)
  - Nested queries (correlation)
  - Ordering
  - Aggregation & Grouping
  - Null values

- Database updates
- Tables and views
- Integrity constraints
Nested Queries

- A **nested query** has another query embedded within it.
  - The embedded query is called the **subquery**.
  - The outer one is called the **outer query**.

```sql
SELECT S.sname
FROM Sailors S
WHERE S.sid IN (SELECT R.sid
                  FROM Reserves R
                  WHERE R.bid = 103 );
```

- The subquery often appears in the **WHERE** clause:
- A subquery can also appear in the **FROM** clause. An example is shown later.
Conceptual Evaluation, extended

- For each row in the cross-product of the outer query, evaluate the WHERE condition by *re-computing the subquery*.

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN ( SELECT R.sid
FROM Reserves R
WHERE R.bid = 103 );
```

equivalent to (can be *simplified* to):

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND R.bid=103;
```
Set Comparison Operators in WHERE

- Set comparisons:
  - `attr IN R` -- true if `R` contains `attr`
  - `EXISTS R` -- true if `R` is non-empty
  - `UNIQUE R` -- true if no duplicates in `R`
  - Any of the above comparators with a proceeding `NOT`

- Set comparisons using a comparator `op` `{<,<=,=,<>, >=,>}`
  - `attr op ALL R` -- every element of `R` satisfies condition
  - `attr op ANY R` -- some element of `R` satisfies condition

  `'attr IN R'` equivalent to  `'attr = ANY R'`
  `'attr NOT IN R'` equivalent to  `'attr <> ALL R'`
Find sid’s of sailors who’ve reserved a red and a green boat

- **INTERSECT**: computes the intersection of any two *union-compatible* sets of tuples.

```sql
SELECT DISTINCT R.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
     AND B.color=‘red’
INTERSECT
SELECT DISTINCT R.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
     AND B.color=‘green’;
```
Simulating INTERSECT

SELECT DISTINCT R.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
AND B.color='red'
AND R.sid IN (SELECT DISTINCT R.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
AND B.color='green');
Find sid’s of sailors who’ve reserved a green boat but not a red boat

- **EXCEPT** computes set difference

```sql
SELECT DISTINCT S.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
    AND B.color=‘green’
EXCEPT
SELECT DISTINCT S.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
    AND B.color=‘red’;
```
Simulating EXCEPT (set difference)

```
SELECT DISTINCT R.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
    AND B.color='green'
    AND R.sid NOT IN (  
        SELECT DISTINCT R.sid
        FROM Reserves R, Boats B
        WHERE R.bid=B.bid
            AND B.color='red'
    );
```
Finding Extreme Values

- Find the sailors with the *highest* rating

```sql
SELECT S.sid
FROM Sailors S
WHERE S.rating >= ALL (SELECT S2.rating
                        FROM Sailors S2);
```
Correlated Subqueries

- A subquery that depends on tables mentioned in the outer query is a **correlated subquery**.
- In conceptual evaluation, must re-compute subquery for each row of the outer query.

```sql
SELECT S.sname
FROM Sailors S
WHERE EXISTS ( SELECT *
FROM Reserves R
WHERE R.bid = 103
AND R.sid = S.sid );
```
Find the names of sailors who’ve reserved all boats

\[
\{ X_{\text{sname}} \mid \exists X_{\text{sid}}, X_{\text{rating}}, X_{\text{age}} \langle X_{\text{sid}}, X_{\text{sname}}, X_{\text{rating}}, X_{\text{age}} \rangle \in \text{Sailors} \wedge \\
\forall \langle X_{\text{bid}}, X_{\text{color}} \rangle \in \text{Boats} \\
( \exists X_{\text{day}} \langle X_{\text{sid}}, X_{\text{bid}}, X_{\text{day}} \rangle \in \text{Reserves} ) \} 
\]
Find the names of 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));

SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS ((SELECT B.bid
FROM Boats B)
EXCEPT
(SELECT R.bid
FROM Reserves R
WHERE R.bid=B.bid
AND R.sid=S.sid));
SQL Overview

- Query capabilities
  - SELECT-FROM-WHERE blocks
  - Set operations (union, intersect, except)
  - Nested queries (correlation)
  - Ordering
    - Aggregation & Grouping
    - Null values
- Database updates
- Tables and views
- Integrity constraints
ORDER BY

- Return the name and age of sailors rated level 8 or above in increasing (decreasing) order of age.

```sql
SELECT S.sname, S.age
FROM Sailors S
WHERE S.rating > 8
ORDER BY S.age [ASC | DESC];
```
TOP-K Queries

- Return the name and age of the ten youngest sailors rated level 8 or above.

```sql
SELECT S.sname, S.age
FROM Sailors S
WHERE S.rating >= 8
ORDER BY S.age ASC
LIMIT 10;
```
SQL Overview

- Query capabilities
  - SELECT-FROM-WHERE blocks
  - Set operations (union, intersect, except)
  - Nested queries (correlation)
  - Ordering
    - Aggregation and grouping
  - Null values
- Database updates
- Tables and views
- Integrity constraints
Aggregate Operators

- Aggregate functions take a relation (single column or multiple columns), and return a value.

\[
\text{SELECT } \text{target-list} \\
\text{FROM } \text{relation-list} \\
\text{WHERE } \text{qualification};
\]

Pass a relation to SELECT.

\[
\text{SELECT } \text{Aggr(attr)} \\
\text{FROM } \text{relation-list} \\
\text{WHERE } \text{qualification};
\]

Convert a relation to a value.

- Aggregate functions take a relation (single column or multiple columns), and return a value.
Example Aggregate Operators

- \[
    \text{SELECT } \text{COUNT}(*)
    \text{ FROM } \text{Sailors} \ S;
\]

- \[
    \text{SELECT } \text{AVG} (S.\text{age})
    \text{ FROM } \text{Sailors} \ S
    \text{ WHERE } \text{S.rating}=10;
\]

- \[
    \text{SELECT } \text{AVG(DISTINCT S.\text{age})}
    \text{ FROM } \text{Sailors} \ S
    \text{ WHERE } \text{S.rating}=10;
\]

- \[
    \text{SELECT } \text{COUNT} (\text{DISTINCT S.rating})
    \text{ FROM } \text{Sailors} \ S
    \text{ WHERE } \text{S.sname}= \text{‘Bob’};
\]

- \[
    \text{SELECT } \text{S.sname}
    \text{ FROM } \text{Sailors} \ S
    \text{ WHERE } \text{S.rating}=
        (\text{SELECT } \text{MAX(S2.rating)}
        \text{ FROM } \text{Sailors} \ S2);
\]
Aggregate Operators

- \( \text{COUNT} (*) \)
- \( \text{COUNT} ([\text{DISTINCT}] A) \)
- \( \text{SUM} ([\text{DISTINCT}] A) \)
- \( \text{AVG} ([\text{DISTINCT}] A) \)
- \( \text{MAX} (A) \)
- \( \text{MIN} (A) \)

- Take a relation (single column or multiple columns), return a value.
- Significant extension of original relational algebra.
Find name and age of the oldest sailor(s)

- The first query is illegal! (We’ll look into the reason more when we discuss `GROUP BY`.)

```sql
SELECT S.sname, MAX (S.age) FROM Sailors S;
```

```sql
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age >= ALL
  (SELECT S2.age
   FROM Sailors S2);
```

```sql
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age =
  (SELECT MAX (S2.age)
   FROM Sailors S2);
```
Motivation for Grouping

- What if we want to apply aggregate operators to each group (subset) of tuples?

- Find the age of the youngest sailor for each rating level.
  - If we know that rating values $\in [1, 10]$, write 10 queries like:
    
    ```
    SELECT MIN (S.age) 
    FROM Sailors S 
    WHERE S.rating = i 
    
    For $i = 1, 2, \ldots, 10$:
    ```
  - In general, we don’t know how many rating levels exist, and what the rating values for these levels are!
Queries With GROUP BY and HAVING

A group is a set of tuples that have the same value for all attributes in grouping-list.

Query returns a single answer tuple for each group!

The target-list can only contain:
(i) attributes that have a single value for a group (e.g., S.rating), or
(ii) aggregate operations on other attributes, e.g., MIN (S.age).
Conceptual Evaluation, extended

- The cross-product of `relation-list` is computed.
- Tuples that fail `qualification` are discarded.
- The remaining tuples are partitioned into groups by the value of attributes in `grouping-list`.
- The `group-qualification`, if present, eliminates some groups.
  - `Group-qualification` must have a single value per group!
- A single answer tuple is produced for each qualifying group.
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors

### SQL Query

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

### Sailors Instance

<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>
<tr>
<td>96</td>
<td>frodo</td>
<td>3</td>
<td>25.5</td>
</tr>
</tbody>
</table>

### Answer Relation

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
Find age of the youngest sailor with age $\geq 18$, for each rating with at least 2 such sailors.

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>1</td>
<td>33.0</td>
</tr>
<tr>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>16.0</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</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>3</td>
<td>25.5</td>
</tr>
<tr>
<td>3</td>
<td>63.5</td>
</tr>
<tr>
<td>3</td>
<td>25.5</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>8</td>
<td>25.5</td>
</tr>
<tr>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>minage</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>25.5</td>
</tr>
<tr>
<td>7</td>
<td>35.0</td>
</tr>
<tr>
<td>8</td>
<td>25.5</td>
</tr>
</tbody>
</table>
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);

- Derived table: result of an SQL query as input to the FROM clause of another query
  - Computed once before the other query is evaluated.
SQL Overview

- Query capabilities
  - SELECT-FROM-WHERE blocks
  - Set operations (union, intersect, except)
  - Nested queries (correlation)
  - Aggregation & Grouping
  - Ordering
  - Null values

- Database updates
- Tables and views
- Integrity constraints
**NULLS in SQL**

- Whenever we don’t have a value, put a **NULL**.
  - Value does not exist.
  - Value exists but is unknown.
  - This attribute is not applicable.
- The schema specifies for each attribute whether it can be null (e.g., **NOT NULL**).
- How does SQL cope with tables that have NULLs?
Null Values

- If \( x = \text{NULL} \), then \( 4 \times (3-x)/7 \) is still NULL

- If \( x = \text{NULL} \), then \( x = 'Joe' \) is UNKNOWN

- In SQL there are three boolean values:
  - FALSE = 0
  - UNKNOWN = 0.5
  - TRUE = 1
**Boolean Expressions involving NULL**

- $C_1$ AND $C_2 = \min(C_1, C_2)$
- $C_1$ OR $C_2 = \max(C_1, C_2)$
- NOT $C_1 = 1 - C_1$
- Rule in SQL: include only tuples that yield TRUE

```sql
SELECT *
FROM Person
WHERE (age < 25) AND
  (height > 6 OR weight > 190);
```

age = 20
height = NULL
weight = 200
Null Values

- Unexpected behavior:

```sql
SELECT *
FROM Person
WHERE age < 25 OR age >= 25;

John’s age is NULL
```
Null Values

- Can test for NULL explicitly:
  - x IS NULL
  - x IS NOT NULL

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25 OR age IS NULL;
```

- Now it includes all Persons.
SQL Overview

- Query capabilities
  - SELECT-FROM-WHERE blocks
  - Set operations (union, intersect, except)
  - Nested queries (correlation)
  - Aggregation & Grouping
  - Ordering
  - Null values
- Database updates
- Tables and views
- Integrity constraints
Modifying the Database

Three kinds of modifications:
- Insert - create new tuple(s)
- Delete - remove existing tuple(s)
- Update - modify existing tuple(s)

- Sometimes they are all called “updates”.


Insertions

General form:

```
INSERT INTO R(A1, …., An)
VALUES (v1,….., vn);
```

Example: Insert a new sailor to the database:

```
INSERT INTO Sailors(sid, sname, rating, age)
VALUES (3212, ‘Fred’, 9, 44);
```

Can omit attributes; a missing attribute is NULL.
May drop attribute names if give values of all attributes in order.
**Insertions**

Example: Insert *multiple* tuples to Sailors:

```
INSERT INTO Sailors(sid, sname)
    SELECT DISTINCT B.sid, B.name
    FROM    Boaters B
    WHERE   Boaters.rank = 'captain';
```

The query replaces the VALUES keyword.
Deletions

Example: delete all tuples that satisfy a condition

```
DELETE
FROM Sailors
WHERE S.sname = 'Harry';
```

Fact about SQL: there is no way to delete only a single occurrence of a tuple that appears twice in a relation.
Updates

Examples:

```
UPDATE Employees
SET salary = salary * 1.1;
```

```
UPDATE Sailors S
SET S.rating = s.rating + 1
WHERE S.sid IN
    (SELECT sid
     FROM Reserves R
     WHERE R.date = 'Oct, 25');
```
SQL Overview

- Query capabilities
  - SELECT-FROM-WHERE blocks
  - Set operations (union, intersect, except)
  - Nested queries (correlation)
  - Aggregation & Grouping
  - Ordering
  - Null values

- Database updates

- Tables and views

- Integrity constraints
Creating Tables

CREATE TABLE Sailors
(   sid INTEGER,
    sname VARCHAR(50) NOT NULL,
    rating INTEGER,
    age REAL,
    PRIMARY KEY (sid));

CREATE TABLE Boats
(   bid INTEGER,
    bname CHAR(20),
    color CHAR(20),
    PRIMARY KEY (bid)
    UNIQUE (bname));

CREATE TABLE Reserves
(   sid INTEGER,
    bid INTEGER,
    day DATE,
    PRIMARY KEY (sid,bid,day),
    FOREIGN KEY (sid) REFERENCES Sailors (sid)
    ON DELETE NO ACTION ON UPDATE CASCADE
    FOREIGN KEY (bid) REFERENCES Boats (bid)
    ON DELETE SET DEFAULT ON UPDATE CASCADE);
Destroying and Altering Tables

- Destroys the Sailors relation, including schema and data.

- The schema is altered by adding a new field; every tuple in the current instance is extended with a null value in the new field.
Views

- A **view** is like a relation, but we store a **definition**, rather than a set of tuples.

```sql
CREATE VIEW RedBoatLovers (sid, name, bid) AS
    SELECT S.sid, S.sname, B.bid
    FROM Sailors S, Reserves R, Boats B
    AND B.color = 'red';
```

- Views can be dropped using **DROP VIEW** command.
  - **DROP TABLE** if there’s a view on the table? Options available...
Uses of Views

- Security/Privacy
  - Views can be used to present necessary information (or a summary) while *hiding details in underlying relation(s)*.
  - E.g., hiding sailors’ credit card from the boat repair dept.

- Logical data independence
  - User application defined on a view is unchanged when underlying table changes.

- Computational benefits
  - Result of a complex query is frequently used; define a view and materialize the result.
  - Online Analytical Processing (OLAP)
SQL Overview

- Query capabilities
  - SELECT-FROM-WHERE blocks
  - Set operations (union, intersect, except)
  - Nested queries (correlation)
  - Aggregation and grouping
  - Ordering
  - Null values
- Database updates
- Tables and views
  - Integrity constraints
Integrity Constraints (Review)

- Types of *integrity constraints* in SQL:
  - Attribute constraints: domain, **NOT NULL**
  - Key constraints: **PRIMARY KEY**, **UNIQUE**
  - Foreign key constraints: **FOREIGN KEY**
  - General constraints: **CHECK**, **ASSERTION**

- Inserts/deletes/updates that violate IC’s are disallowed.
General Constraints

- Two forms: CHECK (single table constraint) and ASSERTION (multiple-table constraint).

```sql
CREATE TABLE Sailors
    ( sid INTEGER,
      sname CHAR(50),
      rating INTEGER,
      age REAL,
      PRIMARY KEY (sid),
      CHECK ( rating >= 1
              AND rating <= 10));
```
Constraints over Multiple Relations

Number of boats plus number of sailors is < 100:

CREATE ASSERTION smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S) +
(SELECT COUNT (B.bid) FROM Boats B) < 100 );

- ASSERTION is a constraint over both tables; checked whenever one of the table is modified.
Review: Participation Constraints

- **Participation constraint:**
  - Every employee works in at least one dept.
  - Each Dept has at least one employee.

- **Participation + Key constraints:**
  - Every department must have one manager.
CREATE ASSERTION Participation_Employee
CHECK
( (SELECT COUNT (DISTINCT ssn) FROM Employees) = 
(SELECT COUNT (DISTINCT ssn) FROM Works_In));

CREATE ASSERTION Participation_Department
CHECK
( (SELECT COUNT (DISTINCT did) FROM Departments) = 
(SELECT COUNT (DISTINCT did) FROM Works_In));
Using Triggers Instead

- A trigger is associated with a table, and activates when an event (insert, delete, update) occurs to a table
- Used to check values or perform computation

```sql
mysql> CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));
Query OK, 0 rows affected (0.03 sec)

mysql> CREATE TRIGGER ins_sum BEFORE INSERT ON account
    -> FOR EACH ROW SET @sum = @sum + NEW.amount;
Query OK, 0 rows affected (0.06 sec)
```

- For full details, see:
  