Structured Query Language

Yanlei Diao
Structured Query Language (SQL)

- **Data Definition Language (DDL)**
  - operating on tables/views

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

- Extension from Relational Algebra / Calculus
  1. From a set to a multi-set (bag) based model
  2. Extending first order expressive power with aggregation and recursion
SQL Overview

- Table definition
- 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
## Example Instances

### Sailor

<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>

### 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>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

### Boats

<table>
<thead>
<tr>
<th>bid</th>
<th>sname</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>green</td>
</tr>
</tbody>
</table>
## Creating Tables

```sql
CREATE TABLE Sailors
(
  sid INTEGER,
  sname CHAR(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.

```sql
DROP TABLE Sailors;
```

- 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.

```sql
ALTER TABLE Sailors
ADD COLUMN credit_card:CHAR(40);
```
SQL Overview

- Table definition
- 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
### 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

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification;
```

- **relation-list**: a list of input relation names, possibly each with a range-variable.

- **qualification**: predicates combined with AND, OR and NOT
  - **predicate**: attr $op$ const or attr1 $op$ attr2, $op$ is $<$, $>$, $\geq$, $\leq$, $=$, $\neq$

- **target-list**: a list of attributes to display in output
  - **DISTINCT** indicates no duplicates in the answer. Default is that duplicates are *not* eliminated!
Conceptual Evaluation Strategy

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification;
```

- **relation-list**: cross-product (\( \times \))
- **qualification**: selection (\( \sigma \))
  - includes join predicates and restrictions on individual tuples
- **target-list**: projection (\( \pi \))
  - duplicate elimination if DISTINCT

This is possibly the least efficient way to execute the query! Leave the issue to Query Optimization…
Example of Conceptual Evaluation

```sql
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>

What is the relational algebra for this query?
Relational Algebra for the Query

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

\[ \pi_{\text{sname}}((\sigma_{\text{bid}=103}\text{Reserves}) \bowtie \text{Sailors}) \]
A Note on Range Variables

- Really needed only if the same relation appears \textit{twice} in the FROM clause.

\begin{verbatim}
SELECT  sname
FROM    Sailors, Reserves
WHERE   Sailors.sid=Reserves.sid
        AND bid=103;
\end{verbatim}

OR

\begin{verbatim}
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND bid=103;
\end{verbatim}

\textbf{It is good style, however, to use range variables always!}
Find sailors who’ve reserved some (at least one) boat

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

Would adding DISTINCT to this query change the answer set?

What if we replace S.sid by S.sname in the SELECT clause and then add DISTINCT?
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 three characters.
- LIKE is used for string matching.
  - ‘_’ stands for any one 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';

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

- Table definition
- 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
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?
  ```sql
  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).
  ```sql
  SELECT DISTINCT S.sid
  FROM Reserves R, Boats B
  WHERE R.bid=B.bid
  AND B.color=‘red’
  UNION
  SELECT DISTINCT S.sid
  FROM Reserves R, Boats B
  WHERE R.bid=B.bid
  AND B.color=‘green’;
  ```

Why do we need DISTINCT in the query?
Set Operations on MultiSets

- Consider a multiset set $A$, for each element, $x \in A$, define an indicator function:
  
  $$1_A(x) = \# \text{ occurrences of } x \text{ in } A$$

- Both regular sets and multisets use consistent definitions:
  
  $$1_{A \cup B}(x) = \max (1_A(x), 1_B(x))$$
  $$1_{A \cap B}(x) = \min (1_A(x), 1_B(x))$$
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 S.sid
FROM Reserves R1, Boats B1,
     Reserves R2, Boats B2
WHERE R1.bid=B1.bid
     AND B1.color=‘red’
     AND B2.color=‘green’;
```

Need **DISTINCT** to be equivalent!

```sql
SELECT DISTINCT S.sid
FROM Reserves R1, Boats B1,
     Reserves R2, Boats B2
     AND (B1.color=‘red’ AND B2.color=‘green’)
     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='red'
EXCEPT
SELECT DISTINCT S.sid
FROM Reserves R, Boats B
WHERE R.bid=B.bid
    AND B.color='green';
```
SQL Overview

- Table definition
- 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
Postgres Setup Instructions

- Visit https://piazza.com/umass/spring2020/cmpsci645/resources

- Download “SetupInstructions.txt” given under the “General” section.

- Follow the steps in the file.

- Test query: Find the names of the sailors who have reserved the boat 103.
**Nested Queries**

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

- The subquery often appears in the **WHERE** clause:

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

- Subqueries are also possible in the **FROM** clause.
Conceptual Evaluation, extended

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

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

However, this query is equivalent to (can be *simplified* to):

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

- A subquery that depends on the table(s) mentioned in the outer query is a **correlated subquery**.
- In conceptual evaluation, must recompute 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 );
```
Set Comparison Operators in WHERE

- Set comparison, optionally with a preceeding NOT:
  - EXISTS R -- true if R is non-empty
  - attr IN R -- true if R contains attr
  - UNIQUE R -- true if no duplicates in R

- Arithmetic operator op \{<,\leq,=,\geq,\rangle\} and ALL/ANY:
  - attr op ALL R -- all elements of R satisfy 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’
```
## Example Instances (Review)

### Sailors1

<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>

### 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>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

### Boats

<table>
<thead>
<tr>
<th>bid</th>
<th>sname</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>green</td>
</tr>
</tbody>
</table>
Finding Extreme Values

- Find the sailors with the highest rating

```sql
SELECT S.sid
FROM Sailors1 S
WHERE S.rating >= ALL ( SELECT S2.rating
                          FROM Sailors1 S2 );
```
Please Write SQL

- Find sailors whose rating is higher than some sailor named Dustin.

```sql
SELECT S.sid
FROM Sailors1 S
WHERE S.rating > ANY (SELECT S2.rating
FROM Sailors1 S2
WHERE S2.sname = 'dustin');
```

- Find sailors whose rating is higher than all sailors named Dustin.

```sql
SELECT S.sid
FROM Sailors1 S
WHERE S.rating > ALL (SELECT S2.rating
FROM Sailors1 S2
WHERE S2.sname = 'dustin');
```
Find sailors who’ve reserved all boats.


(2) 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));
SQL Overview

- Table definition
- 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
Example Aggregate Operators

**Example 1:**
```
SELECT COUNT(*)
FROM Sailors S;
```

**Example 2:**
```
SELECT AVG(S.age)
FROM Sailors S
WHERE S.rating=10;
```

**Example 3:**
```
SELECT AVG(DISTINCT S.age)
FROM Sailors S
WHERE S.rating=10;
```

**Example 4:**
```
SELECT COUNT(DISTINCT S.rating)
FROM Sailors S
WHERE S.sname= 'Bob' ;
```

**Example 5:**
```
SELECT S.sname
FROM Sailors S
WHERE S.rating= (SELECT MAX(S2.rating)
                    FROM Sailors S2);
```
Aggregate Operators

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

- Take a relation (single column or multiple columns), return a value.
- Significant extension of relational algebra.
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.)

```
SELECT S.sname, MAX (S.age) FROM Sailors1 S;
```

```
SELECT S.sname, S.age FROM Sailors1 S WHERE S.age = (SELECT MAX (S2.age) FROM Sailors1 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 \( i \in [1, 10] \), write 10 queries like:

    ```sql
    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

SELECT [DISTINCT] \textit{target-list}
FROM \textit{relation-list}
WHERE \textit{qualification}
GROUP BY \textit{grouping-list}
[HAVING \textit{group-qualification}];

- A \textit{group} is a set of tuples that have the same value for all attributes in \textit{grouping-list}.
- Query returns a \textit{single} answer tuple for each group!
- The \textit{target-list} can only contain:
  (i) attributes in the \textit{grouping-list} (e.g., \textit{S.rating}), or
  (ii) aggregate operations on other attributes, e.g., \textit{MIN} (\textit{S.age}).
Conceptual Evaluation, extended

- The cross-product of \textit{relation-list} is computed.
- Tuples that fail \textit{qualification} are discarded.
- The remaining tuples are partitioned into groups by the value of attributes in \textit{grouping-list}.
- The \textit{group-qualification}, if present, eliminates some groups.
  - \textit{Group-qualification} must have a \textbf{single} value per group!
- A \textbf{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

```
SELECT S.rating, MIN(S.age) AS minage
FROM Sailors2 S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT(*) > 1;
```

**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>

**Sailors2 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>
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>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>
Find those ratings for which the average age is the minimum over all ratings

```sql
SELECT Temp.rating, Temp.avgage
FROM (SELECT S.rating, AVG (S.age) AS avgage
      FROM Sailors1 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

- Table definition
- 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
ORDER BY

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

```
SELECT  S.sname, S.age
FROM    Sailors1 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 Sailors2 S
WHERE S.rating >= 8
ORDER BY S.age ASC
LIMIT 10;
```
SQL Overview

- Table definition
- 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
NULL Values in SQL

- Whenever we don’t have a value, put a NULL.
- Can mean many things:
  - Value does not exist
  - Value exists but is unknown
  - Value 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 \text{NULL}

- If $x = \text{NULL}$, then $x = \text{“Joe”}$ is \text{UNKNOWN}

- In SQL there are three boolean values:
  - \text{FALSE} = 0
  - \text{UNKNOWN} = 0.5
  - \text{TRUE} = 1
Coping with Unknown Values

- C1 AND C2  =  min(C1, C2)
- C1 OR    C2  =  max(C1, C2)
- NOT C1    =  1 - C1

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

E.g.
- age=20
- height=NULL
- weight=200

- Rule in SQL: include only tuples that yield TRUE
Anomaly Associated with Null’s

- Unexpected behavior:

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

E.g. John’s age is NULL

- Some person is not included!
Null Values

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

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

- Now it includes all people.
SQL Overview

- Table definition
- 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:

\[
\text{INSERT INTO R(A1, \ldots, An) VALUES (v1, \ldots, vn);} \]

Example: Insert a new sailor to the database:

\[
\text{INSERT INTO Sailors(sid, sname, rating, age) VALUES (3212, 'Fred', 9, 44);} \]

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 B.id, 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

- Table definition
- 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 CHAR(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
        ON DELETE NO ACTION ON UPDATE CASCADE
    FOREIGN KEY (bid) REFERENCES Boats
        ON DELETE SET DEFAULT ON UPDATE CASCADE);
Destroying and Altering Tables

- Destroys the Sailors relation, including schema and data.

```
DROP TABLE Sailors;
```

- 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.

```
ALTER TABLE Sailors
ADD COLUMN credit_card:CHAR(40);
```
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
WHERE S.sid = R.sid and R.bid = B.bid
    and B.color='red' ;
```

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

- Views can be used to present necessary information (or a summary), while *hiding details in underlying relation(s)*.
- Security/Privacy
  - 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; materialize it.
  - Online Analytical Processing (OLAP)
SQL Overview

- Table definition
- 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.
Questions
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 S.sid
FROM Sailors S, Boats B, Reserves R
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).

```
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 and a green boat

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

```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’;
```

Need **DISTINCT** to be equivalent!

```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’ AND B2.color=‘green’);
```
Find sid’s of sailors who’ve reserved …

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

```sql
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
  AND B.color=‘red’
_except
SELECT S.sid
FROM Sailors S, Boats B, Reserves R
  AND B.color=‘green’;
```
Simulating INTERSECT

- Suppose we have tables $R(a,b)$ and $S(a,b)$
- The following computes $R \cap S$:

```
SELECT DISTINCT *
FROM R
WHERE (R.a, R.b) IN (SELECT *
                     FROM S);
```

This can be expressed without nesting:

- Given $R(a,b)$, $S(a,b)$, what is $R \bowtie S$?

```
SELECT DISTINCT R.a, R.b
FROM R, S
WHERE R.a = S.a AND R.b = S.b;
```

Intersection!
Find the names of sailors who reserved a red and a green boat.

using **INTERSECT**

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

**without INTERSECT (1)**

```sql
SELECT sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
    AND S.sid IN
    (SELECT S2.sid
     FROM Sailors S2, Reserves R2, Boats B2
     WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green')
```

“Find all sailors who have reserved a red boat and, further, have sids that are included in the set of sids of sailors who have reserved a green boat.”
Find the names of sailors who reserved a red and a green boat.

without INTERSECT (2)

```sql
SELECT S1.sname
FROM   (SELECT S.sid, S.name
         FROM Sailors S, Reserves R, Boats B
         WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'green'
         ) AS S1,
       (SELECT S.sid, Sname
         FROM Sailors S, Reserves R, Boats B
         WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
         ) AS S2
WHERE S1.sid = S2.sid AND S1.sname = S2.sname
```
Simulating EXCEPT (set difference)

- What does this query compute?

```sql
SELECT B.bid
FROM    Boats B
WHERE B.bid NOT IN (SELECT R.bid
                     FROM Reserves R
                     WHERE R.sid = 100 );
```

*Find boats not reserved by sailor with sid = 100.*

- Inner on R: boats reserved by sailor with sid=100
- All boats – inner is what we want.
Find those ratings for which the average age is the minimum over all ratings

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.
Integrity Constraints (Review)

- Types of integrity constraints in SQL:
  - Attributes constraints: domain, NOT NULL, CHECK age > 16
  - 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(10),
      rating INTEGER,
      age REAL,
      PRIMARY KEY (sid),
      CHECK ( rating >= 1
              AND rating <= 10));
```
General Constraints (contd.)

- Can use queries to express constraints. Can also name constraints.

```
CREATE TABLE Reserves
  
  ( sname CHAR(10),
    bid INTEGER,
    day DATE,
  PRIMARY KEY (bid, day),
  FOREIGN KEY (bid) REFERENCES Boats
    ON DELETE CASCADE,
  CONSTRAINT noInterlakeRes
    CHECK ('Interlake' <>
      (SELECT B.bname
       FROM Boats B
       WHERE B.bid=bid)));
```
Constraints Over Multiple Relations

- Wrong! Table constraints are required to hold only if the associated table is nonempty.
  - What if Sailors is empty?

- ASSERTION is the right solution; not associated with either table.

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 );

Number of boats plus number of sailors is < 100