SQL Overview

• Query capabilities
  – SELECT-FROM-WHERE blocks,
  – Basic features, ordering, duplicates
  – Set ops (union, intersect, except)
  – Aggregation & Grouping
Aggregation

$$\text{SELECT Avg(S.age) FROM Sailors WHERE S.rating = 10}$$

SQL supports several aggregation operations:

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

```
SELECT  Count(*)
FROM     Sailors
WHERE   rating > 5
```

Except for COUNT, all aggregations apply to a single attribute.
Aggregation: Count

COUNT applies to duplicates, unless otherwise stated:

```
SELECT Count(category)
FROM Product
WHERE year > 1995
```

Better:

```
SELECT Count(DISTINCT category)
FROM Product
WHERE year > 1995
```
Simple Aggregation

Purchase(product, date, price, quantity)

Example 1: find total sales for the entire database

```
SELECT Sum(price * quantity)
FROM Purchase
```

Example 1’: find total sales of bagels

```
SELECT Sum(price * quantity)
FROM Purchase
WHERE product = ‘bagel’
```
GROUP BY and HAVING clauses

• We often want to apply aggregates to each of a number of groups of rows in a relation.

Find the age of the youngest sailor for each rating level.

```
SELECT MIN (S.age)
FROM   Sailors S
WHERE  S.rating = i
```

For i = 1, 2, ... 10
### Sailors

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33</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>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35</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>74</td>
<td>horatio</td>
<td>9</td>
<td>35</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
<tr>
<td>71</td>
<td>zorba</td>
<td>10</td>
<td>16</td>
</tr>
</tbody>
</table>

### New Table

<table>
<thead>
<tr>
<th>rating</th>
<th>age?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
Queries With GROUP BY and HAVING

- 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.
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!

- One answer tuple is generated per qualifying group.
For sailors over 18, find the age of the youngest sailor, for each rating with at least 2 such sailors

```
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>
SQL Overview

• Query capabilities
  – SELECT-FROM-WHERE blocks,
  – Basic features, ordering, duplicates
  – Set ops (union, intersect, except)
  – Aggregation & Grouping
  – Nested queries (correlation)
  – Null values
Nested queries

• A nested query is a query with another query embedded within it.
• The embedded query is called the subquery.
• The subquery usually appears in the WHERE clause:

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

(Subqueries also possible in FROM or HAVING clause.)
Conceptual evaluation, extended

• For each row in cross product of outer query, evaluate the WHERE clause conditions, (re)-computing the subquery.

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

equivalent to:

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

• If the inner subquery depends on tables mentioned in the outer query then it is a **correlated subquery**.

• In terms of conceptual evaluation, we must recompute subquery for each row of outer query.

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS ( SELECT *
               FROM Reserves R
               WHERE R.bid = 103
               AND R.sid = S.sid )
```
Set-comparison operators

• Optional NOT may precede these:
  – EXISTS R -- true if R is non-empty
  – attr IN R -- true if R contains attr
  – UNIQUE R -- true if no duplicates in R

• For arithmetic operator \( \text{op} \) \{\(<,\leq,=,\geq,\geq,>\}\)
  – attr \( \text{op} \) ALL -- all elements of R satisfy condition
  – attr \( \text{op} \) ANY R -- some element of R satisfies condition

**IN** equivalent to \( = \) \text{ANY}
**NOT IN** equivalent to \( <> \) \text{ALL}
Example

• Find the sailors with the highest rating

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

• Find sailors whose rating is higher than some sailor named Horatio.

```
SELECT S.sid
FROM   Sailors S
WHERE  S.rating > ANY (SELECT S2.rating
                        FROM Sailors S2
                        S2.name = 'Horatio')
```

• Find sailors whose rating is higher than all sailors named Horatio.

```
SELECT S.sid
FROM   Sailors S
WHERE  S.rating > ALL (SELECT S2.rating
                          FROM Sailors S2
                          S2.name = 'Horatio')
```
Find boats **not** reserved by sailor with sid = 100.

- B: all boats
- R: boats reserved by sailor with sid=100
- B – R is what we want.

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

• Find the names of sailors who have reserved some boat
• (i.e. there exists a boat they reserved)

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

• Existential conditions are natural and easy.
Universal conditions

• Find the names of sailors who have reserved all boats (i.e. *for each* boat, they have reserved it.)
• Recall: relational division
• Universal conditions are harder.

```
SELECT  S.sname  
FROM     Sailors S  
WHERE NOT EXISTS (  
    Set of boats not reserved by S.sid  
)  
```
Universal conditions

- Find the names of sailors who have reserved all boats.

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS (  
    SELECT B.bid
    FROM Boats B
    WHERE B.bid NOT IN (SELECT R.bid
                          FROM Reserves R
                          WHERE R.sid = S.sid )
)
```

For each sailor, check that there is no boat s/he hasn’t reserved.
Simulating INTERSECT

• Suppose we have tables $R(a,b)$ and $S(a,b)$

• The following computes $R \cap S$:

  ```sql
  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 $R \bowtie S$?

  Intersection!

  ```sql
  SELECT DISTINCT R.a, R.b
  FROM R, S
  WHERE R.a = S.a AND R.b = S.b;
  ```
Find the names of sailors who reserved a red and a green boat.

using INTERSECT

```
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

```
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.”
Simulating EXCEPT (set difference)

• What does this query compute?

```sql
SELECT DISTINCT *
FROM R
WHERE (R.a, R.b) NOT IN (SELECT * FROM S);
```

Can this be expressed without a nested query? No.

(But this fact is not obvious)
SQL Overview

• Query capabilities
  – SELECT-FROM-WHERE blocks,
  – Basic features, ordering, duplicates
  – Set operations (union, intersect, except)
  – Aggregation & Grouping
  – Nested queries (correlation)
  – Null values
NULLS in SQL

• Whenever we don’t have a value, we can put a NULL
• Can mean many things:
  – Value does not exist
  – Value exists but is unknown
  – Value not applicable
  – Etc.
• The schema specifies for each attribute whether it can be null (nullable attribute)
• 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 \( x = \text{“Joe”} \) is UNKNOWN

• In SQL there are three boolean values:
  \begin{align*}
  \text{FALSE} & = 0 \\
  \text{UNKNOWN} & = 0.5 \\
  \text{TRUE} & = 1
  \end{align*}
Null Values

- \( C_1 \text{ AND } C_2 = \min(C_1, C_2) \)
- \( C_1 \text{ OR } C_2 = \max(C_1, C_2) \)
- \( \text{NOT } C_1 = 1 - C_1 \)

Rule in SQL: include only tuples that yield TRUE

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

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

Unexpected behavior:

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

Some Persons are 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 Persons
Outer Joins

• In a typical join, tuples of one relation that don’t match any tuple on join conditions are omitted from result.
• In an outer join, tuples without a match may be preserved in the output.
• Missing values are filled with NULL.
Outer Joins

- **LEFT OUTER JOIN**: rows of *left* relation without matching row in right relation appear in result.
- **RIGHT OUTER JOIN**: rows of *right* relation without matching row in left relation appear in result.
- **FULL OUTER JOIN**: rows of both relations appear in result.
## Outer Joins

### 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</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</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</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>10/10</td>
</tr>
</tbody>
</table>

```sql
SELECT S.sid, R.bid
FROM Sailors S LEFT OUTER JOIN Reserves R
```
SQL Overview

• SQL Preliminaries
• Integrity constraints
• Query capabilities
  – SELECT-FROM-WHERE blocks,
  – Basic features, ordering, duplicates
  – Set ops (union, intersect, except)
  – Aggregation & Grouping
  – Nested queries (correlation)
  – Null values
• Modifying the database
• Views

Review in the textbook, Ch 5
Modifying the Database

Three kinds of modifications
• Insertion - creates new tuple(s)
• Deletion - remove existing tuple(s)
• Updates - modify existing tuple(s)

Sometimes they are all called “updates”
Insertions

General form:

\[
\text{INSERT INTO } R(A_1, \ldots, A_n) \text{ VALUES } (v_1, \ldots, v_n)
\]

Example: Insert a new sailor to the database:

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

Missing attribute → NULL.
May drop attribute names if give them in order.
Insertions

```
INSERT INTO Sailor(sname)
SELECT DISTINCT B.name
FROM Boaters B
WHERE Boaters.rank = "captain"
```

The query replaces the VALUES keyword. Here we insert many tuples into PRODUCT.
Deletions

Example:

```
DELETE
FROM   Sailor
WHERE  S.sname = 'Horatio'
```

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

Example:

```sql
UPDATE   Sailor S
SET    rating = rating + 1
WHERE  Sailor.sid  IN  
                    (SELECT sid
                      FROM   Reserves R
                      WHERE  R.date =ʻOct, 25’);
```
Views
Views

• A **view** is a relation defined by a query.
• The query defining the view is called the **view definition**
• For example:

```sql
CREATE VIEW Developers AS
   SELECT name, project
   FROM Employee
   WHERE department = "Development"
```
Virtual and Materialized Views

- **Two types of views:**
  - **virtual**: the view relation is defined, but not computed or stored.
    - Computed only on-demand – slow at runtime
    - Always up to date
    - In SQL: CREATE VIEW ... AS SELECT
  - **materialized**: the view relation is computed and stored in system.
    - Pre-computed offline – fast at runtime
    - May have stale data
    - In SQL: CREATE TABLE ... AS SELECT
Virtual view example

Person(name, city)
Purchase(buyer, seller, product, store)
Product(name, maker, category)

CREATE VIEW Seattle-view AS

SELECT buyer, seller, product, store
FROM Person, Purchase
WHERE Person.city = "Seattle" AND
     Person.name = Purchase.buyer

We have a new virtual table:
Seattle-view(buyer, seller, product, store)
We can use the view in a query as we would any other relation:

```
SELECT  name, store
FROM      Seattle-view, Product
WHERE   Seattle-view.product = Product.name  AND
                 Product.category = "shoes"
```
Querying a virtual view

```
SELECT name, Seattle-view.store
FROM Seattle-view, Product
WHERE Seattle-view.product = Product.name AND
      Product.category = "shoes"
```

"View expansion"

```
SELECT name, Purchase.store
FROM Person, Purchase, Product
WHERE Person.city = "Seattle" AND
      Person.name = Purchase.buyer AND
      Purchase.product = Product.name AND
      Product.category = "shoes"
```
The great utility of views

• Data independence
• Efficient query processing
  - materializing certain results can improve query execution
• Controlling access
  - Grant access to views only to filter data
• Data integration
  - Combine data sources using views
View-related issues

1. View selection
   • which views to materialize, given workload

2. View maintenance
   • when base relations change, (materialized) views need to be refreshed.

3. Updating virtual views
   • can users update relations that don’t exist?

4. Answering queries using views
   • when only views are available, what queries over base relations are answerable?
View Maintenance

• Two steps:
  ▪ Propagate: Compute changes to view when data changes.
  ▪ Refresh: Apply changes to the materialized view table.

• Maintenance policy: Controls when we do refresh.
  ▪ Immediate: As part of the transaction that modifies the underlying data tables. (+ Materialized view is always consistent; - updates are slowed)
  ▪ Deferred: Some time later, in a separate transaction. (- View becomes inconsistent; + can scale to maintain many views without slowing updates)
Deferred Maintenance

• Three flavors:
  ▪ **Lazy**: Delay refresh until next query on view; then refresh before answering the query.
  ▪ **Periodic (Snapshot)**: Refresh periodically. Queries possibly answered using outdated version of view tuples. Widely used, especially for asynchronous replication in distributed databases, and for warehouse applications.
  ▪ **Event-based**: E.g., Refresh after a fixed number of updates to underlying data tables.
Updating Views

How can I insert a tuple into a table that doesn’t exist?

Employee(ssn, name, department, project, salary)

CREATE VIEW Developers AS
   SELECT name, project
   FROM Employee
   WHERE department = "Development"

If we make the following insertion:

INSERT INTO Developers VALUES("Joe", "Optimizer")

It becomes:

INSERT INTO Employee(ssn, name, department, project, salary) VALUES(NULL, "Joe", "Development", "Optimizer", NULL)
Non-Updatable Views

Person(name, city)
Purchase(buyer, seller, product, store)

CREATE VIEW City-Store AS

SELECT Person.city, Purchase.store
FROM Person, Purchase
WHERE Person.name = Purchase.buyer

How can we add the following tuple to the view?

(“Seattle”, “Nine West”)

We don’t know the name of the person who made the purchase; cannot set to NULL.
Troublesome examples

CREATE VIEW OldEmployees AS
   SELECT name, age
   FROM Employee
   WHERE age > 30

INSERT INTO OldEmployees VALUES("Joe", 28)

If this tuple is inserted into view, it won’t appear. Allowed by default in SQL!
Ambiguous updates

<table>
<thead>
<tr>
<th>Name</th>
<th>group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>fac</td>
</tr>
<tr>
<td>Bob</td>
<td>fac</td>
</tr>
<tr>
<td>Bob</td>
<td>cvs</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>group</th>
<th>file</th>
</tr>
</thead>
<tbody>
<tr>
<td>fac</td>
<td>foo.txt</td>
</tr>
<tr>
<td>fac</td>
<td>bar.txt</td>
</tr>
<tr>
<td>cvs</td>
<td>foo.txt</td>
</tr>
</tbody>
</table>

Join

Delete ("Alice", "foo.txt")

view

<table>
<thead>
<tr>
<th>Alice</th>
<th>foo.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>bar.txt</td>
</tr>
<tr>
<td>Bob</td>
<td>foo.txt</td>
</tr>
<tr>
<td>Bob</td>
<td>bar.txt</td>
</tr>
</tbody>
</table>
Updating views in practice

• Updates on views highly constrained:
  – SQL-92: updates only allowed on single-table views with projection, selection, no aggregates.
  – SQL-99: takes into account primary keys; updates on multiple table views may be allowed.
  – SQL-99: distinguishes between updatable and insertable views