CS W4111.001
Introduction to Databases
Fall 2019

Computer Science Department
Columbia University
SQL: DDL and DML

• Data Definition Language (DDL): subset of SQL to create, modify, and delete tables, with their integrity constraints (earlier lectures)

• Data Manipulation Language (DML): subset of SQL to insert, delete, and modify tuples, as well as to query the tables (now)
Relations for Examples

- **Sailors** *(sid, sname, rating, age)*
- **Boats** *(bid, bname, color)*
- **Reserves** *(sid, bid, day)*

Reserves keeps track of which sailors reserved which boats and when
### Example Instances

#### Reserves

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/2018</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/2018</td>
</tr>
</tbody>
</table>

#### 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>

Boats also has an instance, not shown here.
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification

- **relation-list**: List of relation names (possibly with a range variable after each name)
- **target-list**: List of attributes of relations in relation-list
- **qualification**: Comparisons `attribute_1 op constant` or `attribute_1 op attribute_2`, where `op` is one of `<`, `>`, `<=`, `>=`, `=`, `<>` (or `!=`), combined using AND, OR, and NOT
- **DISTINCT**: Optional keyword that indicates that the answer should not contain duplicates; by default, if DISTINCT is not included, duplicates are not eliminated (note difference with relational algebra)
Relational Algebra vs. SQL

• $\sigma_{\text{rating}>7}(\text{Sailors})$: SELECT *
  FROM Sailors S
  WHERE S.rating>7

• Sailors $\times$ Reserves: SELECT *
  FROM Sailors, Reserves

• $\sigma_{\text{age}>40}(\text{Sailors})$ $\times$ Reserves: SELECT *
  FROM Sailors S, Reserves R
  WHERE S.age>40

• $\Pi_{\text{sname}}(\sigma_{\text{rating}>7}(\text{Sailors}))$: SELECT S.sname
  FROM Sailors S
  WHERE S.rating>7
Basic SQL Query

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

- relation-list: List of relation names (possibly with a range variable after each name)
- target-list: List of attributes of relations in relation-list
- qualification: Comparisons attribute op constant or attribute$_1$ op attribute$_2$, where op is one of $<$, $>$, $\leq$, $\geq$, $=$, $\neq$ (or $!=$), combined using AND, OR, and NOT
- DISTINCT: Optional keyword that indicates that the answer should not contain duplicates; by default, if DISTINCT is not included, duplicates are not eliminated (note difference with relational algebra)
Conceptual Evaluation Strategy of SQL Query

Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:

1. Compute the cross-product of the relations in relation-list in FROM clause
2. Discard cross-product tuples that do not satisfy the qualification in WHERE clause
3. Delete attributes that are not in target-list in SELECT clause
4. If DISTINCT is specified, eliminate duplicate rows

This strategy is probably the least efficient way to answer a query, because of Step 1; the query optimizer will find efficient strategies to compute the same answers.
### 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>10/10/2018</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/2018</td>
</tr>
</tbody>
</table>

```sql
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND R.bid=103
```
Range Variables in SQL Queries

• Really needed only if same relation appears twice in FROM clause (or elsewhere in query, as we will see)
• However, it is good style to use range variables always, for clarity and brevity:

```sql
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid=R.sid AND R.bid=103
S and R are range variables here, for Sailors and Reserves, respectively
```
Range Variables in SQL Queries

Movie(title, year, length, filmType, studioName)
MovieStar(title, year, starName)

“Find titles that have been used for 2 or more movies”

SELECT Old.title
FROM Movie Old, Movie New
WHERE Old.title=New.title AND Old.year<New.year
“Find sids of Sailors Who Have Reserved at Least One Boat”

SELECT R.sid
FROM Reserves R

Would adding DISTINCT to this query make a difference?

• Yes!
• Without DISTINCT, duplicate tuples are not eliminated from query results, and duplicates are indeed possible for this query, if a sailor has made multiple reservations
“List title, year, and studioName of All Movies with Meryl Streep or Amy Adams”

SELECT M.title, M.year, M.studioName
FROM Movie M, MovieStar S
WHERE M.title=S.title AND M.year=S.year AND
  (S.starName='Meryl Streep' OR S.starName='Amy Adams')

Would DISTINCT make a difference here?
Arithmetic Expressions, String Pattern Matching

“For each sailor whose name begins and ends with B, and contains at least three characters, return the sailor’s age, the age minus 5, and twice the age”

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

- **AS** and **=** are two ways to name attributes in result
- Output has three attributes: age, age1, and age2
- **LIKE** is used for string matching: ‘_’ stands for exactly one character; ‘%’ stands for 0 or more characters
UNION of 2 Union-Compatible Tables

“Find sids of sailors who have reserved a red or a green boat”

• Without UNION:
  SELECT R.sid
  FROM Boats B, Reserves R
  WHERE B.bid=R.bid AND (B.color='red' OR B.color='green')

• With UNION:
  SELECT R.sid
  FROM Boats B, Reserves R
  WHERE B.bid=R.bid AND B.color='red'
  UNION
  SELECT R.sid
  FROM Boats B, Reserves R
  WHERE B.bid=R.bid AND B.color='green'
INTERSECT of 2 Union-Compatible Tables

“Find sids of sailors who have reserved a red and a green boat”

• Without INTERSECT:
  SELECT R1.sid
  FROM Boats B1, Reserves R1, Boats B2, Reserves R2
  WHERE B1.bid=R1.bid AND B1.color='red' AND
  B2.bid=R2.bid AND B2.color='green' AND
  R1.sid=R2.sid

• With INTERSECT:
  SELECT R.sid
  FROM Boats B, Reserves R
  WHERE B.bid=R.bid AND B.color='red'
  INTERSECT
  SELECT R.sid
  FROM Boats B, Reserves R
  WHERE B.bid=R.bid AND B.color='green'
EXCEPT (= Difference) of 2 Union-Compatable Tables

“Find sids of sailors who have reserved a red boat but not a green boat”

SELECT R.sid
FROM Boats B, Reserves R
WHERE B.bid=R.bid AND B.color='red'
EXCEPT
SELECT R.sid
FROM Boats B, Reserves R
WHERE B.bid=R.bid AND B.color='green'
UNION, INTERSECT, EXCEPT: Duplicates?

• In general, duplicates in SQL are not eliminated unless the DISTINCT keyword is added next to SELECT

• However, the default behavior is different for UNION, INTERSECT, and EXCEPT:
  • By default, duplicates are indeed eliminated in the results of UNION, INTERSECT, and EXCEPT: we first eliminate duplicates from the two relations and then apply UNION (with duplicates eliminated from result), INTERSECT, and EXCEPT as in relational algebra, with set semantics
  • To retain duplicates, with bag semantics, use UNION ALL, INTERSECT ALL, and EXCEPT ALL
Duplicates in UNION ALL, INTERSECT ALL, and EXCEPT ALL

Assume relation \textbf{R} has \textit{m} copies of tuple t and relation \textbf{S}, which is union compatible with \textbf{R}, \textbf{has n copies} of the same tuple t.

Then, tuple t appears:

- \text{m+n} times in \textbf{R UNION ALL S}
- \text{min\{m, n\}} times in \textbf{R INTERSECT ALL S}
- \text{max\{0, m-n\}} times in \textbf{R EXCEPT ALL S}
Nested Queries: Queries Within Queries

• A very powerful feature of SQL: WHERE clause can itself contain a query:

  “Find names of sailors who have reserved boat #103”
  
  SELECT S.sname
  FROM Sailors S
  WHERE S.sid IN (SELECT R.sid
                    FROM Reserves R
                    WHERE R.bid=103)

• To understand semantics of nested queries, think of conceptual evaluation strategy: for each Sailors tuple, check qualification by computing subquery

• To find sailors who have not reserved boat #103, use NOT IN instead of IN
“IN” Might Involve Several Attributes

WHERE (S.sid, S.sname) IN (SELECT S2.sid, S2.sname FROM Sailors S2 WHERE ...)

...
Nested Queries with Correlation

“Find names of sailors who reserved boat #103”

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

- EXISTS is true when applied to a nonempty relation
- The results of the correlated nested query depend on the Sailors tuple that is being considered at each time, so the nested query must be recomputed for each Sailors tuple
Nested Queries with Correlation

“Find names of sailors who reserved boat #103 at most once”

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

- UNIQUE is true when applied to a relation with no duplicates; UNIQUE is true in particular over an empty relation
- Why did we have to replace * by `R.bid` in SELECT clause of subquery?
More on Set-Comparison Operators

• IN, EXISTS, and UNIQUE can be used with negation: NOT IN, NOT EXISTS, and NOT UNIQUE

• Additional operators: \( op \) ANY, \( op \) ALL, where \( op \) is \(<, >, \leq, \geq, =, \neq \) (or \(!=\) )

“Find sailors whose rating is greater than that of some sailor named ‘Horatio’”
SELECT *
FROM Sailors S
WHERE S.rating > ANY (SELECT S2.rating
 FROM Sailors S2
 WHERE S2.sname=‘Horatio’)
“Find Titles that Have Been Used for 2 or More Movies” (revisited)

• Without ANY:
  SELECT Old.title
  FROM Movie Old, Movie New
  WHERE Old.title=New.title AND Old.year<New.year

• With ANY:
  SELECT Old.title
  FROM Movie Old
  WHERE Old.year < ANY(SELECT New.year
                                                        FROM Movie New
                                                        WHERE New.title=Old.title)
“Find Those Sailors With the Highest Rating”

```sql
SELECT S.sid
FROM Sailors S
WHERE S.rating >= ALL(SELECT S2.rating
FROM Sailors S2)
```
Aggregate Operators: A Significant Extension of Relational Algebra

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

Example SQL queries:

- `SELECT COUNT (*) FROM Sailors S`  
- `SELECT AVG (S.age) FROM Sailors S WHERE S.rating=10`  
- `SELECT AVG (DISTINCT S.age) FROM Sailors S WHERE S.rating=10`  
- `SELECT S.sname FROM Sailors S WHERE S.rating = (SELECT MAX(S2.rating) FROM Sailors S2)`
“Find name and age of Oldest Sailors”

• Illegal query:
  SELECT S.sname, MAX (S.age)
  FROM Sailors S
  (Not well defined: it intends to return the sname of each tuple in Sailors, together with the MAX age computed over all tuples in Sailors.)

• Correct query:
  SELECT S.sname, S.age
  FROM Sailors S
  WHERE S.age = (SELECT MAX (S2.age)
                 FROM Sailors S2)
More Examples

• “Find number of distinct students enrolled in Fall 2019 courses”; use Enrolled(sid, cid, semester)

```
SELECT COUNT(DISTINCT E.sid)
FROM Enrolled E
WHERE E.semester='Fall 2019'
```

• “Find average number of movies made per studio in 2018”

```
SELECT COUNT(*)/COUNT(DISTINCT M.studioName)
FROM Movie M
WHERE M.year=2018
```
“Find the age of the youngest sailor for each rating level”

- If we knew that rating values ranged from 1 to 10, we could write 10 separate queries, for i=1, 2, ..., 10:
  ```sql
  SELECT MIN (S.age) 
  FROM Sailors S 
  WHERE S.rating = i 
  ```

- This is tedious, plus we might not even know how many rating levels exist, or what the rating values for these levels are
“Find the age of the youngest sailor for each rating level”

SELECT S.rating, MIN (S.age)
FROM Sailors S
GROUP BY S.rating

This query (1) groups Sailors tuples by rating, so that we have one group with all tuples with rating=1, another group with all tuples with rating=2, and so on; (2) “collapses” each group into exactly one tuple in the query results, consisting of the rating of the group (which is well defined, because we created the groups by rating) and the minimum age for the tuples in the group.
“For Each Studio, Find the Sum of the Lengths of All of its Movies”

SELECT M.studioName, SUM(M.length)
FROM Movie M
GROUP BY M.studioName
What is this query equivalent to?

```sql
SELECT M.studioName
FROM Movie M
GROUP BY M.studioName
```
Queries With GROUP BY and HAVING Clauses

- target-list in SELECT clause contains (i) attribute names and/or (ii) terms with aggregate operations
- Attribute list (i) in target-list must be equal to or a subset of the grouping-list attributes in GROUP BY clause
  Intuitively, each answer tuple corresponds to a group, and these attributes must have a single value per group
- group-qualification in HAVING clause follows analogous requirements as target-list in SELECT clause
  (group-qualification condition must have a single value per group)
Conceptual Evaluation Expanded

1. Compute the cross-product of the relations in relation-list in FROM clause
2. Discard cross-product tuples that do not satisfy the qualification in WHERE clause
3. Partition remaining tuples into groups by the value of attributes in grouping-list in GROUP BY clause
4. Discard groups that do not satisfy the group-qualification in HAVING clause
5. Delete attributes that are not in target-list in SELECT clause
6. If DISTINCT is specified, eliminate duplicate rows
"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 SELECT, GROUP BY, or HAVING clauses; other attributes not needed
• Second 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</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</td>
</tr>
<tr>
<td>64</td>
<td>horatio</td>
<td>7</td>
<td>35</td>
</tr>
<tr>
<td>29</td>
<td>brutus</td>
<td>1</td>
<td>33</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>35</td>
</tr>
</tbody>
</table>
“For Each Red Boat, Find the Number of Reservations for this Boat”

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

- Grouping over a join of two relations
- What do we get if we remove B.color='red' from the WHERE clause and add a HAVING clause with this condition?

Illegal! B.color not in GROUP BY clause nor in an aggregate
“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 S2.rating=S.rating)

• HAVING clause can also contain a subquery
• Compare this query with the earlier query where we considered only ratings with 2 sailors 18 or older
• What if we replace HAVING clause with HAVING COUNT(*) >1? Not OK: according to conceptual evaluation, WHERE clause applied before grouping …
NULL Values

Possible interpretations of a NULL value:

- Value unknown (e.g., unknown age of a sailor: the sailor’s age exists, but we don’t know it)
- Value inapplicable (e.g., spouse name for a single person)
- Value withheld (e.g., a private phone number of an employee)

- NULL is like a regular value in some sense
- But also NULL is unlike a regular value (e.g., NULL is not equal to NULL)
NULL Behaves Differently In Many Contexts

• If NULL is involved in an operation, result is NULL
  If x is NULL, then x*2 is also NULL

• If NULL is involved in a comparison, result is UNKNOWN
  If x is NULL, then the comparison x>2 is UNKNOWN

• NULL is not a constant
  It is illegal to have a comparison x=NULL; instead, we write x IS NULL (or x IS NOT NULL)

• NULL in duplicate elimination?
  NULL values are considered equal for duplicate elimination
NULL and Aggregate Operators

• Most aggregate operators ignore NULL values
  So COUNT (A) is the number of tuples with non-NULL values for attribute A

• However, COUNT (*) counts them

• Also, a query:
  SELECT R.A, AVG (R.B)
  FROM R
  GROUPBY R.A
  will produce exactly one tuple with NULL for the value of R.A if such value exists in R, with the average value for R.B for the tuples with R.A equal to NULL
NULL Leads to 3-Valued Logic

• Conditions can evaluate to TRUE, FALSE, or UNKNOWN
• A condition in the WHERE clause of a query is only satisfied for tuples for which it evaluates to TRUE (not FALSE or UNKNOWN); same thing for HAVING
• To reason about UNKNOWN, think of TRUE=1, FALSE=0, and UNKNOWN=0.5; then AND=min, OR=max, and NOT=1-
  • TRUE AND UNKNOWN is UNKNOWN
  • TRUE OR UNKNOWN IS TRUE
  • NOT UNKNOWN is UNKNOWN, and so on
What Does this Query Return?

SELECT *
FROM Sailors S
WHERE S.age>=18 OR S.age<18