Plan for Today

- Enhance understanding of semantics of conceptual query evaluation
- Build on understanding of the role of primary keys and NULL values in queries
- Practice reading and writing more complex SQL queries

Relational Calculus

- Tuple Relational Calculus:
  - Variables range over (i.e., get bound to) tuples
  - **Answer tuples**: an assignment of constants to variables that make an expression evaluate to true

\[ \{ S \mid S \in \text{Sailors} \land S.\text{rating} > 7 \} \]

\[ \{ P \mid \exists S \in \text{Sailors}(S.\text{rating} > 7 \land P.\text{name} = S.\text{name} \land P.\text{Page} = S.\text{age}) \} \]

- Every relational algebra query can be expressed as a **safe** calculus query, and vice versa

Check out Section 4.3 in the book for more!

Rel. Alg. Compound Operator: Division

- Useful for expressing “for all” queries like:
  *Find names of sailors who have reserved all boats.*

- For A/B, attributes of B are subset of attributes of A
  - May need to use “project” operator first
  - E.g., let A have 2 fields, x and y; B has only field y:

\[ A/B = \{ \langle x \rangle \mid \forall \langle y \rangle \in B(\exists \langle x, y \rangle \in A) \} \]

\( A/B \) contains all \( x \) tuples such that for every \( y \) tuple in \( B \), there exists a tuple \( x, y \) in \( A \)

Infinite!

This is unsafe.

---

CS 133: Databases

Fall 2016
Lec 8 – 9/26

Prof. Beth Trushkowsky
SQL: Structured Query Language

- Relational algebra and calculus form the basis for SQL
- The standard query language supported by most commercial DBMS
- Specification: originally IBM, then ANSI starting 1986
  - IBM System R
  - ANSI SQL 89
  - ANSI SQL 92
  - ANSI SQL 99
  - ANSI SQL 2003
  - ANSI SQL 2008
  - ANSI SQL 2011

Query Semantics

- Semantics of an SQL query are defined in terms of the following conceptual evaluation strategy:
  1. do FROM clause: compute cross-product of tables (e.g., Students and Enrolled).
  2. do WHERE clause: Check conditions, discard tuples that fail. (i.e., “selection”).
  3. do SELECT clause: Delete unwanted fields. (i.e., “projection”).
  4. If DISTINCT specified, eliminate duplicate rows.

Not necessarily an efficient way to compute a query!
  - An optimizer will find more efficient strategies to get the same answer.

Basic SQL Query

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

- **Relation-list**: A list of relation names
  - possibly with a range-variable after each name
- **Target-list**: A list of attributes of tables in relation-list
  - possibly also expressions
  - can be replaced by “*” if you want all attributes in result
- **Qualification**: Comparisons combined using AND, OR and NOT.
  - Comparisons are Attr op const or Attr1 op Attr2, where op is one of =, <>, <, >, ≤, ≥
- **DISTINCT**: optional keyword indicating that the answer should not contain duplicates.
  - in SQL SELECT clause, the default is that duplicates are not eliminated!

Helpful for looking at results: ORDER BY field(s) [ASC|DESC], LIMIT num_rows

Example Relation Instances

<table>
<thead>
<tr>
<th>sid</th>
<th>surname</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>

**Sailors**

<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>

**Reserves**

<table>
<thead>
<tr>
<th>bid</th>
<th>bname</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Interlake</td>
<td>blue</td>
</tr>
<tr>
<td>102</td>
<td>Interlake</td>
<td>red</td>
</tr>
<tr>
<td>103</td>
<td>Clipper</td>
<td>green</td>
</tr>
<tr>
<td>104</td>
<td>Marine</td>
<td>red</td>
</tr>
</tbody>
</table>

**Boats**

(Assume appropriate foreign key constraints are used)
Visualizing Query Evaluation

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

Range Variables

- Can associate "range variables" with the relations in the FROM clause
  - saves writing, makes queries easier to understand
  - like an alias

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

- Needed when ambiguity could arise
  - for example, if same relation used multiple times in same FROM clause (called a "self-join")

Range Variables (cntd)

- Example where range variables are required (self-join example):

```sql
SELECT S1.sname, S1.age, S2.sname, S2.age
FROM Sailors S1, Sailors S2
WHERE S1.age = S2.age
    AND S1.rating > S2.rating;
```

- Could the result contain a pair of Sailors that are actually the same person?

Null Values

- Field values in a tuple are sometimes missing
  - unknown (e.g., a rating or grade has not been assigned)
  - inapplicable (e.g., no spouse’s name).
  - SQL provides a special value `null` for such situations.

- The presence of `null` complicates things. E.g.:
  - Is "rating > 8" true or false when rating is null?
  - What about AND, OR and NOT?
  - Check if a value is/is not `null` using `IS NULL`
Null Values – 3 Valued Logic

We need a 3-valued logic.
- Values: True, False and Unknown
- Meaning of constructs must be defined carefully
  (e.g., WHERE clause eliminates rows that do not evaluate to true.)

<table>
<thead>
<tr>
<th>AND</th>
<th>T</th>
<th>F</th>
<th>Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>Unknown</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>NULL</td>
<td>Unknown</td>
<td>F</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>T</th>
<th>F</th>
<th>Null</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>Unknown</td>
</tr>
<tr>
<td>NULL</td>
<td>T</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

Expressions

- Can use arithmetic expressions in SELECT clause
  (plus other calculations we’ll discuss later)
- Use AS to provide column names

```sql
SELECT S.sname, S.rating % 2 AS evenOrOddRating
FROM Sailors S
WHERE S.age >= 18;
```

- Can also have expressions in WHERE clause:

```sql
SELECT S1.sname AS name1, S2.sname AS name2
FROM Sailors S1, Sailors S2
WHERE 2*S1.rating > S2.rating;
```

String operations

- SQL also supports some string operations
- “LIKE” is used for string matching.

```sql
SELECT S.sname, S.rating
FROM Sailors S
WHERE S.sname LIKE 'B_%B'
```

‘_’ stands for any one character and ‘%’ stands for 0 or more arbitrary characters.

Query: Find sids of sailors who’ve reserved a red or a green boat

```sql
SELECT DISTINCT R.sid
FROM Boats B, Reserves R
WHERE R.bid = B.bid
  AND (B.color = 'red' OR B.color = 'green');
```

UNION: compute the union of any two union-compatible sets of tuples (which are themselves the result of SQL queries)

```sql
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';
```

Why do we want DISTINCT here?
Query: Find sids of sailors who’ve reserved a red \textbf{and} a green boat

- If we simply replace \texttt{OR} by \texttt{AND} in the previous query, we get the wrong answer. (Why?)

```
SELECT DISTINCT R.sid
FROM Boats B, Reserves R
WHERE R.bid = B.bid
AND (B.color = 'red' \texttt{AND} B.color = 'green')
```

red \textbf{and} a green boat (cntd)...

- \texttt{INTERSECT}:
  - Discussed in textbook.
  - Can be used to compute the intersection of any two \textit{union-compatible} sets of tuples.

- Also in textbook: \texttt{EXCEPT} (sometimes called MINUS)
  - Included in the SQL 92 standard,
  - but many systems don’t support them.

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

red \textbf{and} a green boat (cntd)...

- What’s wrong with this version of the previous query?

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

Nested Queries

- Can use SQL queries to aid the evaluation of another SQL query

- \texttt{WHERE} clause can itself contain an SQL query!
  - Actually, so can \texttt{FROM} and \texttt{HAVING} clauses.

- Example:

```
SELECT S.sid
FROM Sailors S
WHERE S.rating > (SELECT AVG(rating) FROM Sailors);
```

How many results does this subquery return?
Nested Queries

- Subqueries can also be relations with many tuples

**Names of sailors who’ve reserved boat #103:**

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

- Semantics of nested queries:
  - Think of a nested loops evaluation: For each Sailors tuple, check the qualification by computing the subquery

- To find sailors who have not reserved #103, use NOT IN

  In general, watch out for attributes that could be NULL!

Nested Queries with Correlation

**Find names of sailors who’ve reserved boat #103:**

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

- Subquery recomputed for each Sailors tuple.
  - Think of subquery as a function call that runs a query!

- What if we replaced EXISTS with UNIQUE, and replaced SELECT * with SELECT R.bid?

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

More on Set-Comparison Operators

- SQL operators to filter tuples by applying to a relation R to get a boolean result
  - `value IN R`: true iff value is equal to one of the values in unary R
  - `EXISTS R`: true iff R is not empty
  - `UNIQUE R`: true iff R has no duplicates (or is empty)
  - `value <op> ANY R`: true iff value <op> some value in unary R
  - `value <op> ALL R`: true iff value <op> all values in unary R

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

Rewriting INTERSECT Queries Using IN

**Find sids of sailors who’ve reserved both a red and a green boat:**

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

- Similarly, EXCEPT queries can be re-written using NOT IN.

- **Exercise 5**: How would you change this query to find names (not sids) of Sailors who have reserved both red and green boats?
What does this query do?

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS (SELECT B.bid
                 FROM Boats B
                 WHERE B.color = 'green'
                 AND NOT EXISTS (SELECT R.bid
                                  FROM Reserves R
                                  WHERE R.bid = B.bid
                                  AND R.sid = S.sid));
```

*Names of Sailors S such that…*

…there is no green boat B…

...without a reservation between this sailor and that boat

*Division!* ➔ Names of sailors who have reserved all green boats