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 \textit{Sailors} \land S.\textit{rating} > 7 \} \}

  \{ P \mid \exists S \in \textit{Sailors}(S.\textit{rating} > 7 \land \textit{Pname} = S.\textit{sname} \land \textit{Page} = S.\textit{age}) \}

  Effectively the projected attributes

- 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!
Logical Query Plan Example

- Example: college database
  - Students(SID, name, gpa)
  - Enrolled(SID, CID, grade)

SELECT S.name, E.CID
FROM Students S, Enrolled E
WHERE S.SID=E.SID;

relational algebra expression?

Sets of tuples flow upward

Get tuples from Students

Get tuples from Enrolled

Combine

Pull out name and CID fields

Basic SQL Query

SELECT [DISTINCT] target-list
FROM relation-list
[WHERE qualification]
ORDER BY field(s) [ASC|DESC]
LIMIT num_rows

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.
**Visualizing Query Evaluation**

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

Join condition: are these the same sid?
Is this bid 103?

**Example Relation Instances**

We will use these instances of relations in our examples.

<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>95</td>
<td>Bob</td>
<td>3</td>
<td>63.5</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>95</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

(Any tuple)

**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

Expressions

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

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

- Can also have expressions in WHERE clause:

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

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>Unknown</td>
<td>Unknown</td>
<td>Unknown</td>
</tr>
</tbody>
</table>

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

```
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)

```
(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';
```
Query: Find sids of sailors who’ve reserved a red and a green boat

- If we simply replace OR by 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' AND B.color = 'green')
```

red and a green boat (cntd)...

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

- Also in textbook: 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 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

  WHERE clause can itself contain an SQL query!
  - Actually, so can FROM and 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`. 
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));

Division! → Names of sailors who have reserved all green boats