Announcements

• HW1 reminder:
  – Due on 09/21 (Thurs), 11:55 pm, no late days

• Project proposal reminder:
  – Due on 09/20 (Wed), 11:55 pm by email

• Your piazza, sakai, gradiance accounts should be active
  – Occasional Pop up quizzes will start from next week
  – Bring a laptop in class
  – Part of class participation

Recap: Lecture 2

• XML overview
  – differences with relational model and transformation

• SQL
  – Creating/modifying relations
  – Specifying integrity constraints
  – Key/candidate key, superkey, primary key, foreign key

Today’s topic

• More SQL
  – semantic
  – joins
  – group bys and aggregates
  – nested queries

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Basic SQL Query

```
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 relations in relation-list

• qualification Comparisons
  – (Attr op const) or (Attr1 op Attr2)
  – where op is one of <, >, ≤, ≥ combined using and or not

• DISTINCT is an optional keyword indicating that the answer should not contain duplicates
  – Default is that duplicates are not eliminated

Conceptual Evaluation Strategy

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

• Semantics of an SQL query defined in terms of the following conceptual evaluation strategy:
  – Compute the cross-product of <relation-list>
  – Discard resulting tuples if they fail <qualifications>
  – Delete attributes that are not in <target-list>
  – If distinct is specified, eliminate duplicate rows

• This strategy is probably the least efficient way to compute a query!
  – An optimizer will find more efficient strategies to compute the same answers
Example of Conceptual Evaluation

**Step 1:** Form cross product of Sailor and Reserves

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

**Step 2:** Discard tuples that do not satisfy qualification

```sql
WHERE S.sid=R.sid
```

**Step 3:** Select the specified attribute(s)

```sql
S.sid sname rating age
22 dustin 7 45
31 lubber 8 55
58 rusty 10 35
```

Example of Conceptual Evaluation

**A Note on “Range Variables”**

- Really needed only if the same relation appears twice in the from clause
  - sometimes used as a short-name
- The previous query can also be written as:

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

OR

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

It is good style, however, to use range variables always!

Example of Conceptual Evaluation

**Find sailor ids who’ve reserved at least one boat**

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

Example of Conceptual Evaluation

**Find sailor ids who’ve reserved at least one boat**

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

- Would adding DISTINCT to this query make a difference?
Joins

• Condition/Theta-Join
• Equi-Join
• Natural-Join
• (Left/Right/Full) Outer-Join

Condition/Theta Join

```
SELECT * 
FROM Sailors S, Reserves R 
WHERE S.age=R.age and age = 40
```

Form cross product, discard rows that do not satisfy the condition

```
sid    name    rating    age
22      dustin  7         45
22      dustin  7         58
31      lubber  8         55
58      rusty   10        35
```

Equi Join

```
SELECT * 
FROM Sailors S, Reserves R 
WHERE S.age=R.age 
```

A special case of theta join
Join condition only has equality predicate =

```
sid    name    rating    age    sid    bid    day
22      dustin  7         45       22      101    10/10/96
22      dustin  7         58       22      103    11/12/96
31      lubber  8         55       31      101    10/10/96
31      lubber  8         58       31      103    11/12/96
58      rusty   10        35       58      101    10/10/96
58      rusty   10        35       58      103    11/12/96
```

Natural Join

```
SELECT * 
FROM Sailors S NATURAL JOIN Reserves R 
```

A special case of equi join
Equality condition on ALL common predicates (sid)
Duplicate columns are eliminated

```
sid    name    rating    age    sid    bid    day
22      dustin  7         45       22      101    10/10/96
22      dustin  7         58       22      103    11/12/96
31      lubber  8         55       31      101    10/10/96
31      lubber  8         58       31      103    11/12/96
58      rusty   10        35       58      101    10/10/96
58      rusty   10        35       58      103    11/12/96
```

Outer Join

```
SELECT S.sid, R.bid 
FROM Sailors S LEFT OUTER JOIN Reserves R 
ON S.Sid=R.Sid
```

Preserves all tuples from the left table whether or not there is a match
If no match, fill attributes from right with null
Similarly RIGHT/FULL outer join

```
sid    bid
22      101
31      null
58      103
```

Expressions and Strings

```
SELECT S.age, age1=S.age-5, 2% age AS age2 
FROM Sailors S 
WHERE S.name LIKE 'B_%'
```

• Illustrates use of arithmetic expressions and string pattern matching
• Find triples (of ages of sailors and two fields defined by expressions) for sailors
  — whose names begin and end with B and contain at least three characters
• LIKE is used for string matching. _% stands for any one character and %_ stands for 0 or more arbitrary characters
  — You will need these often
Find sid's of sailors who've reserved a red or a green boat

• Assume a Boats relation
• UNION: Can be used to compute the union of any two union-compatible sets of tuples
  — can themselves be the result of SQL queries
• If we replace OR by AND in the first version, what do we get?
• Also available: EXCEPT (What do we get if we replace UNION by EXCEPT?)

```
SELECT S.sid FROM Sailors S, Boats B, Reserves R
WHERE ...
UNION
SELECT S.sid FROM Sailors S, Boats B, Reserves R
WHERE ...
```

Find names of sailors who've reserved boat #103:

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

• EXISTS is another set comparison operator, like IN
• Illustrates why, in general, subquery must be re-computed for each Sailors tuple

Nested Queries with Correlation

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

• If UNIQUE is used, and * is replaced by R.bid, finds sailors with at most one reservation for boat #103
  — UNIQUE checks for duplicate tuples
More on Set-Comparison Operators

- We’ve already seen IN, EXISTS and UNIQUE
- Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: op ANY, op ALL, op IN
  - where op : >, ≤, =, ≥
- Find sailors whose rating is greater than that of some sailor called Horatio
  - similarly ALL

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

Aggregate Operators

Check yourself: What do these queries compute?

```
SELECT COUNT (?) FROM Sailors S
SELECT AVG (S.age) FROM Sailors S WHERE S.rating=10
SELECT COUNT (DISTINCT S.rating) FROM Sailors S WHERE S.sname='Bob'
SELECT AVG (DISTINCT S.age) FROM Sailors S WHERE S.rating=10
```

Motivation for Grouping

- So far, we’ve applied aggregate operators to all (qualifying) tuples
  - Sometimes, we want to apply them to each of several groups of tuples
- Consider: Find the age of the youngest sailor for each rating level
  - In general, we don’t know how many rating levels exist, and what the rating values for these levels are!
  - Suppose we know that rating values go from 1 to 10; we can write 10 queries that look like this (need to replace i by num):
    - awesome: SELECT MIN (S.age) FROM Sailors S WHERE S.rating = i
    - for i = 1, 2, …, 10

```
For i = 1, 2, …, 10
SELECT MIN (S.age) FROM Sailors S WHERE S.rating = i
```

Conceptual Evaluation

- The cross-product of relation-list is computed
- Tuples that fail qualification are discarded
- ‘Unnecessary’ fields are deleted
- 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
    - In effect, an attribute in group-qualification that is not an argument of an aggregate op also appears in grouping list
    - like “…GROUP BY bid, sid HAVING bid = 3”
- One answer tuple is generated per qualifying group

```
SELECT S.rating, MIN (S.age) AS minage FROM Sailors S WHERE S.age >= 18
```

Queries With GROUP BY and HAVING

```
SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
```

- The target-list contains
  - (i) attribute names
  - (ii) terms with aggregate operations (e.g., MIN (S.age))
- The attribute list () 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
  - Here a group is a set of tuples that have the same value for all attributes in grouping-list

```
First go over the examples in the following slides
Then come back to this slide and study yourself
```

Find age of the youngest sailor with age >= 18, 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
```

```
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>
<tr>
<td>9</td>
<td>33.5</td>
</tr>
<tr>
<td>10</td>
<td>45.0</td>
</tr>
<tr>
<td>30</td>
<td>95.0</td>
</tr>
<tr>
<td>74</td>
<td>35.0</td>
</tr>
<tr>
<td>85</td>
<td>25.5</td>
</tr>
<tr>
<td>95</td>
<td>63.5</td>
</tr>
<tr>
<td>96</td>
<td>25.5</td>
</tr>
</tbody>
</table>
```
SQL provides a special value null for such situations.

Null Values

- Field values in a tuple are sometimes
  - unknown, e.g., a rating has not been assigned, or
  - inapplicable, e.g., no spouse's name
- SQL provides a special value null for such situations.
Standard Boolean 2-valued logic

- True = 1, False = 0
- Suppose X = 5
  - (X > 100) AND (X >= 1) is ...
  - (X > 100) OR (X >= 1) is ...
  - NOT(X = 5) is ...
- Intuitively,
  - T = 1, F = 0
  - For V1, V2 ∈ {1, 0}
  - V1 ∧ V2 = MIN(V1, V2)
  - V1 ∨ V2 = MAX(V1, V2)
  - ¬(V1) = 1 – V1

2-valued logic does not work for nulls

- Suppose rating = null, X = 5
- Is rating > 8 true or false?
- What about AND, OR and NOT connectives?
  - (rating > 8) AND (X = 5)?
- What if we have such a condition in the WHERE clause?

3-Valued Logic For Null

- TRUE (= 1), FALSE (= 0), UNKNOWN (= 0.5)
  - unknown is treated as 0.5
- Now you can apply rules from 2-valued logic!
  - For V1, V2 ∈ {1, 0, 0.5}
  - V1 ∧ V2 = MIN(V1, V2)
  - V1 ∨ V2 = MAX(V1, V2)
  - ¬(V1) = 1 – V1
- Therefore,
  - NOT UNKNOWN = UNKNOWN
  - UNKNOWN OR TRUE = TRUE
  - UNKNOWN AND TRUE = UNKNOWN
  - UNKNOWN AND FALSE = FALSE
  - UNKNOWN OR FALSE = UNKNOWN

New issues for Null

- The presence of NULL complicates many issues. E.g.:
  - Special operators needed to check if value IS/IS NOT NULL
  - Be careful!
  - WHERE X = NULL does not work!
  - Need to write "WHERE X IS NULL"
- Meaning of constructs must be defined carefully
  - e.g., WHERE clause eliminates rows that don’t evaluate to true
  - So not only FALSE, but UNKNOWNS are eliminated too
  - very important to remember!
- But NULL allows new operators (e.g. outer joins)
- Arithmetic with NULL
  - all of +, *, / return null if any argument is null
- Can force “no nulls” while creating a table
  - sname char(20) NOT NULL
  - primary key is always not null

Aggregates with NULL

- What do you get for
  - SELECT count(*) from R1?
  - SELECT count(rating) from R1?

Aggregates with NULL

- What do you get for
  - SELECT count(*) from R1?
  - SELECT count(rating) from R1?
Aggregates with NULL

- What do you get for
  - SELECT count(*) from R1?
  - SELECT count(rating) from R1?

- What do you get for
  - SELECT count(*) from R2?
  - SELECT count(rating) from R2?

Overview: General Constraints

- Useful when more general ICs than keys are involved
- There are also ASSERTIONS to specify constraints that span across multiple tables
- There are TRIGGERS too: procedure that starts automatically if specified changes occur to the DBMS
  - see additional slides at the end

Can create a new table from a query on other tables too

SELECT … INTO … FROM … WHERE

CREATE TABLE Reserves
  ( sname CHAR(10),
    bid INTEGER,
    day DATE,
    PRIMARY KEY (bid,day),
    CONSTRAINT noInterlakeRes
      CHECK (`Interlake' NOT IN
        (SELECT B.bname
         FROM Boats B
         WHERE B.bid=bid)))

CREATE TABLE Sailors
  ( sid INTEGER,
    sname CHAR(10),
    rating INTEGER,
    age REAL,
    PRIMARY KEY (sid),
    CHECK ( rating >= 1 AND rating <= 10 )

CREATE TABLE Students
  ( sid INTEGER,
    sname CHAR(10),
    age REAL,
    PRIMARY KEY (sid),
    CHECK ( rating >= 1 AND rating <= 10 )

CREATE VIEW YoungActiveStudents
  (name, grade)
AS
  SELECT S.name, E.grade
  FROM Students S, Enrolled E
  WHERE S.sid = E.sid and S.age<21

Views

- A view is just a relation, but we store a definition, rather than a set of tuples

CREATE VIEW YoungActiveStudents
  (name, grade)
AS
  SELECT S.name, E.grade
  FROM Students S, Enrolled E
  WHERE S.sid = E.sid and S.age<21

- Views can be dropped using the DROP VIEW command

- Views and Security: Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s)
  - the above view hides courses "cid" from E
“WITH” clause – very useful!

- You will find “WITH” clause very useful!

```sql
WITH Temp1 AS
(SELECT ....),
Temp2 AS
(SELECT ....
SELECT X, Y
FROM TEMP1, TEMP2
WHERE....
```

- Can simplify complex nested queries

Summary

- SQL has a huge number of constructs and possibilities
  - You need to learn and practice it on your own
  - Given a problem, you should be able to write a SQL query and verify whether a given one is correct
- Pay attention to NULLs