### Outer Joins in Relational Algebra

- If a tuple in R doesn’t match any tuple in S, then it won’t be “represented” in $R \bowtie_\sigma S$:

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>S</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>y</td>
<td>y</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Outer joins include such tuples in result, so natural left outer join of R and S is:

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>x</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>y</td>
<td>i</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>z</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Joins in SQL Revisited

SELECT [DISTINCT] target_list
FROM
table1 [INNER | {LEFT|RIGHT|FULL} {OUTER}] JOIN table2
ON qualification_list
WHERE ...

• INNER is default
• Difference in how to deal with NULL values

PostgreSQL documentation:

Inner Join, Natural Join

SELECT S.sid, S.name, R.bid
FROM Sailors S, Reserves R
WHERE S.sid = R.sid

SELECT S.sid, S.name, R.bid
FROM Sailors S INNER JOIN Reserves R
ON S.sid = R.sid

SELECT S.sid, S.name, R.bid
FROM Sailors S NATURAL JOIN Reserves R

Natural join means equijoin for each pair of attributes with same name
Find sailor names and their reserved boat ids

```
SELECT S.sid, S.name, R.bid
FROM Sailors S INNER JOIN Reserves R
    ON S.sid = R.sid
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eugene</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
<td>8</td>
<td>27</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>102</td>
<td>9/12</td>
</tr>
<tr>
<td>2</td>
<td>102</td>
<td>9/13</td>
</tr>
</tbody>
</table>

Query result (note no tuple for Ken!)

```
<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eugene</td>
<td>102</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>102</td>
</tr>
</tbody>
</table>
```

**Left Outer Join**

- Returns all matched rows and all unmatched rows from table on left of join clause
  
  So at least one row for each row in left table…

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

- All sailors who have reserved boats appear with the corresponding bid’s
- All sailors who have reserved no boats also appear but with bid set to NULL
Left Outer Join

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

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eugene</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
<td>8</td>
<td>27</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>102</td>
<td>9/12</td>
</tr>
<tr>
<td>2</td>
<td>102</td>
<td>9/13</td>
</tr>
</tbody>
</table>

Query result (we now have a tuple for Ken!)

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eugene</td>
<td>102</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>102</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Right Outer Join

Same as LEFT OUTER JOIN, but guarantees results for rows in table on right side of JOIN

```
SELECT S.sid, S.name, R.bid
FROM Reserves R RIGHT OUTER JOIN Sailors S
    ON R.sid = S.sid
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eugene</td>
<td>102</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>102</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Full Outer Join

Returns all matched or unmatched rows from both sides of JOIN

```
SELECT S.sid, S.name, R.bid
FROM Sailors S FULL OUTER JOIN Reserves R
ON S.sid = R.sid
```

<table>
<thead>
<tr>
<th>Sailors</th>
<th>Reserves</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>1</td>
<td>Eugene</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
</tr>
</tbody>
</table>

Query result

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>bid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eugene</td>
<td>102</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>102</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>109</td>
</tr>
</tbody>
</table>

Why is sid NULL?

(ignore for this example the fact that Reserves violates referential integrity)
ORDER BY, LIMIT

SELECT [DISTINCT] target-list
FROM relation-list
WHERE qualification
GROUP BY grouping-list
HAVING group-qualification
ORDER BY order-list
LIMIT limit-expr [OFFSET offset-expr]

ORDER BY

SELECT S.name, S.rating, S.age
FROM Sailors S
ORDER BY S.rating ASC, S.age DESC

List of order-list expressions dictates ordering precedence:
- Sort primarily in ascending order by rating
- If there are ties on rating, sort them in descending order by age
- If there are ties on both rating and age, sort them arbitrarily
ORDER BY

(:: means type "cast")

SELECT S.name, (S.rating/2)::INTEGER AS rat2, S.age
FROM Sailors S
ORDER BY (S.rating/2)::INTEGER ASC, S.age DESC

- Default is ASC
- Could add NULLS FIRST or NULLS LAST to indicate position of NULL in order for attribute in ORDER BY
- By default, NULL sorts as if it were larger than non-NULL values

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eugene</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
<td>8</td>
<td>27</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>rat2</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Luis</td>
<td>1</td>
<td>39</td>
</tr>
<tr>
<td>Ken</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td>Eugene</td>
<td>4</td>
<td>22</td>
</tr>
</tbody>
</table>

ORDER BY

SELECT S.name, (S.rating/2)::INTEGER AS rat2, S.age
FROM Sailors S
ORDER BY (S.rating/2)::INTEGER ASC, S.age ASC

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eugene</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
<td>8</td>
<td>27</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>rat2</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Luis</td>
<td>1</td>
<td>39</td>
</tr>
<tr>
<td>Eugene</td>
<td>4</td>
<td>22</td>
</tr>
<tr>
<td>Ken</td>
<td>4</td>
<td>27</td>
</tr>
</tbody>
</table>
**LIMIT**

```sql
SELECT S.name, (S.rating/2)::INTEGER AS rat2, S.age
FROM Sailors S
ORDER BY (S.rating/2)::INTEGER ASC, S.age DESC
LIMIT 2
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eugene</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
<td>8</td>
<td>27</td>
</tr>
</tbody>
</table>

**Sailors**

<table>
<thead>
<tr>
<th></th>
<th>Sailors</th>
<th>Query result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>1</td>
<td>Eugene</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
<td>8</td>
</tr>
</tbody>
</table>

Includes only the top-2 tuples (LIMIT 2)

- To have predictable results, LIMIT should always be used with ORDER BY
- LIMIT ALL and LIMIT NULL are equivalent to omitting LIMIT clause

**LIMIT with OFFSET**

```sql
SELECT S.name, (S.rating/2)::INTEGER AS rat2, S.age
FROM Sailors S
ORDER BY (S.rating/2)::INTEGER ASC, S.age DESC
LIMIT 2 OFFSET 1
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eugene</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
<td>8</td>
<td>27</td>
</tr>
</tbody>
</table>

**Sailors**

<table>
<thead>
<tr>
<th></th>
<th>Sailors</th>
<th>Query result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>1</td>
<td>Eugene</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
<td>8</td>
</tr>
</tbody>
</table>

Includes only the top-2 tuples (LIMIT 2) after skipping the top tuple (OFFSET 1)
LIMIT with OFFSET

SELECT S.name, (S.rating/2)::INTEGER AS rat2, S.age
FROM Sailors S
ORDER BY (S.rating/2)::INTEGER ASC, S.age DESC
LIMIT (SELECT COUNT(*) / 2
FROM Sailors AS S2)

← can have expressions instead of constants

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eugene</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
<td>8</td>
<td>27</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>rat2</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Luis</td>
<td>1</td>
<td>39</td>
</tr>
</tbody>
</table>

WITH: “Defining Tables” Just for a Query

WITH tablename(attr1, ...) AS (select_query)
    [, tablename(attr1, ...) AS (select_query)]
main_select_query
WITH

WITH RedBoats(bid, count) AS
  (SELECT B.bid, COUNT(*)
   FROM Boats B, Reserves R
   WHERE B.bid = R.bid AND B.color = 'red'
   GROUP BY B.bid)
SELECT B.bname, RB.count
FROM Boats AS B, RedBoats AS RB
WHERE B.bid = RB.bid AND RB.count < 2

Names of unpopular red boats, with their number of reservations

Views: Defining “Tables” in Terms of Other Tables

CREATE VIEW <view_name>
AS <select_statement>

• “Tables” defined as query results rather than through inserting base data
  • Helpful to make development simpler
  • Helpful for security
• At query time, references to view_name replaced with select_statement
• Similar to WITH but persistent, not associated with just one query
Defining a view for popular boats

CREATE VIEW boat_counts
AS
SELECT R.bid, COUNT(*)
FROM Reserves R
GROUP BY R.bid
HAVING COUNT(*) > 10

SELECT B.bname
FROM boat_counts BC,
Boats B
WHERE B.bid = BC.bid

Query to find names of popular boats, expressed using view

Rewritten expanded query

CREATE TABLE with Query

CREATE TABLE <table_name> AS
<select_statement>

• Schema of table is inherited from SELECT but can be overridden:

CREATE TABLE boats_jane1 AS
SELECT R.bid
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND S.sname='Jane'

boats_jane1(bid INTEGER)

CREATE TABLE boats_jane2 AS
SELECT R.bid AS foo
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND S.sname='Jane'

boats_jane2(foo INTEGER)

• How is this different than views? (Hint: What if we insert a new tuple into Reserves?)
Modifying a Relation Schema

- ALTER TABLE Sailors ADD spouse_id INTEGER;
  spouse_id attribute initialized as NULL for existing Sailors
- ALTER TABLE Sailors ADD phone CHAR(12) DEFAULT 'unlisted';
  In contrast, phone attribute initialized with a default value 'unlisted'
- ALTER TABLE Sailors DROP age;
  Modifications succeed only if they don’t conflict with the rest of the existing schema (e.g., cannot drop an attribute that is part of a primary key, for example)

Modifying a Relation Schema

- ALTER TABLE Sailors DROP CONSTRAINT SidIsKey;
  Can also drop constraints, but they have to have a name
- Can add constraints as well, but must be valid at the time they are added
- Can also “CASCADE CONSTRAINTS” (e.g., to drop all foreign keys that refer to the primary key attributes that you are dropping)
Event-Condition-Action Rules: Triggers

- Only awakened when a certain **event** happens (e.g., insert, delete, update)
- A trigger tests a **condition** when an event awakens it; if trigger condition is false, nothing happens
- If trigger condition is true, the associated **action** is performed by the DBMS (e.g., prevent the event from happening or undo the effects of the event); action can be any sequence of database operations!

Triggers in SQL: Options

- A condition may be specified in WHEN clause: action executed only if the rule is triggered and the condition holds when triggering event occurs
- Action executed either before or after the triggering event
- Action can refer to both old and/or new values of tuples that were inserted, deleted, or updated in the event that triggered the action
- Action is performed either:
  - Once for each modified tuple, or
  - Once for all tuples changed in one database operation
Trigger to foil any attempt to lower the net worth of a movie executive
MovieExec(name, address, cert#, netWorth) relation

CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON MovieExec
REFERENCING OLD ROW AS OldTuple,
    NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
    UPDATE MovieExec
    SET netWorth=OldTuple.netWorth
    WHERE cert#=NewTuple.cert#;

Trigger to prevent average net worth of movie executives from dropping below $500K
MovieExec(name, address, cert#, netWorth) relation

CREATE TRIGGER AvgNetWorthTrigger
AFTER UPDATE OF netWorth ON MovieExec
REFERENCING OLD TABLE AS OldStuff,
    NEW TABLE AS NewStuff
FOR EACH STATEMENT
WHEN (500000 > (SELECT AVG(M.netWorth) FROM MovieExec M))
BEGIN
    DELETE FROM MovieExec
    WHERE (name, address, cert#, netWorth) IN NewStuff;
    INSERT INTO MovieExec
        (SELECT * FROM OldStuff)
END;
User Defined Functions (UDFs)

- Custom functions that can be called in database
- Many languages: SQL, Python, C, Perl, …

```sql
CREATE FUNCTION function_name(p1 type, p2 type, …)
RETURNS type
AS $$
-- logic
$$
$$ LANGUAGE language_name;
```

A Simple UDF, Written in SQL

CREATE FUNCTION mult1(v INTEGER) RETURNS INTEGER
AS $$
SELECT v * 10;
$$ LANGUAGE SQL;

```sql
SELECT mult1(S.age) AS age10
FROM Sailors AS S
```

<table>
<thead>
<tr>
<th>Sid</th>
<th>Name</th>
<th>Rating</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eugene</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>2</td>
<td>39</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
<td>8</td>
<td>27</td>
</tr>
</tbody>
</table>

Result:

<table>
<thead>
<tr>
<th>age10</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>220</td>
<td></td>
</tr>
<tr>
<td>390</td>
<td></td>
</tr>
<tr>
<td>270</td>
<td></td>
</tr>
</tbody>
</table>

https://www.postgresql.org/docs/9.3/static/xfunc-sql.html
Another UDF Written in SQL, with Tuple as Input

CREATE FUNCTION mult2(x Sailors) RETURNS INTEGER AS $$
SELECT (x.sid + x.age) / x.rating;
$$ LANGUAGE SQL;

SELECT mult2(*) AS age3
FROM Sailors AS S

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>rating</th>
<th>age</th>
<th>age3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Eugene</td>
<td>7</td>
<td>22</td>
<td>3.286</td>
</tr>
<tr>
<td>2</td>
<td>Luis</td>
<td>2</td>
<td>39</td>
<td>20.5</td>
</tr>
<tr>
<td>3</td>
<td>Ken</td>
<td>8</td>
<td>27</td>
<td>3.75</td>
</tr>
</tbody>
</table>

Procedural Language/SQL, or PLSQL

• Extension of SQL with instructions common in programming languages (IF ELSE statements, etc.)

CREATE FUNCTION proc(v INTEGER) RETURNS INTEGER AS $$
DECLARE
    -- define variables
BEGIN
    -- PL/SQL code
END;
$$ LANGUAGE plpgsql;

https://www.postgresql.org/docs/9.3/static/plpgsql.html
Procedural Language/SQL

CREATE FUNCTION proc(v INTEGER) RETURNS INTEGER AS $$
DECLARE
    qty INTEGER = 10;
BEGIN
    qty = qty * v;
    INSERT INTO blah VALUES(qty);
    RETURN qty + 2;
END;
$$ LANGUAGE plpgsql;

Procedural Code in Python 2, or plpython2u (u="untrusted")

CREATE FUNCTION proc(v INTEGER) RETURNS INTEGER AS $$
import random
return random.randint(0, 100) * v
$$ LANGUAGE plpython2u;

• Very powerful: can do anything so must be careful; run in a Python interpreter with no security protection
• plpy Python module provides database access (e.g., plpy.execute("select 1"))

http://www.postgresql.org/docs/9.3/static/plpython.html
Procedural Code in Python 2, or plpython2u (u="untrusted")

CREATE FUNCTION proc(v TEXT) RETURNS TEXT
AS $$
import requests
resp = requests.get('http://google.com/search?q=%s' % v)
return resp.content.decode('unicode-escape')
$$ LANGUAGE plpython2u;