Outer Joins
The joins you know from RA

These can go in a FROM clause, or can be stand-alone queries:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>R, S</td>
<td>R × S</td>
</tr>
<tr>
<td>R cross join S</td>
<td>R × S</td>
</tr>
<tr>
<td>R natural join S</td>
<td>R ⋈ S</td>
</tr>
<tr>
<td>R join S on Condition</td>
<td>R ⋈_{condition} S</td>
</tr>
</tbody>
</table>
In practise natural join is dangerous

• A working query can be broken by adding a column to a schema.
  
  • Example:
    
    SELECT sID, instructor
    FROM Student NATURAL JOIN Took
    NATURAL JOIN Offering;
  
  • What if we add a column called campus to Offering?
  
  • Also, having implicit comparisons impairs readability.
  
  • Best practise: Don’t use natural join.
Dangling tuples

• With joins that require some attributes to match, tuples lacking a match are left out of the results.
• We say that they are “dangling”.
• An outer join preserves dangling tuples by padding them with \textbf{NULL} in the other relation.
• A join that doesn’t pad with \textbf{NULL} is called an inner join.
Three kinds of outer join

• **LEFT OUTER JOIN**
  • Preserves dangling tuples from the relation on the LHS by padding with nulls on the RHS.

• **RIGHT OUTER JOIN**
  • The reverse.

• **FULL OUTER JOIN**
  • Does both.
Example: joining R and S various ways

R

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

S

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

R NATURAL JOIN S

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Example

R NATURAL FULL JOIN S

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Example

R NATURAL LEFT JOIN S

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Example

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

**R NATURAL RIGHT JOIN S**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>NULL</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
Summary of join expressions

Cartesian product

\[ A \ \text{CROSS JOIN} \ B \]

same as \( A, B \)

Theta-join

\[ A \ \text{JOIN} \ B \ \text{ON} \ C \]

✓\( A \ \{\text{LEFT} | \text{RIGHT} | \text{FULL}\} \ \text{JOIN} \ B \ \text{ON} \ C \)

Natural join

\[ A \ \text{NATURAL JOIN} \ B \]

✓\( A \ \text{NATURAL} \ \{\text{LEFT} | \text{RIGHT} | \text{FULL}\} \ \text{JOIN} \ B \ \text{ON} \ C \)

✓ indicates that tuples are padded when needed.
Keywords INNER and OUTER

• There are keywords INNER and OUTER, but you never need to use them.
• Your intentions are clear anyway:
  • You get an outer join iff you use the keywords LEFT, RIGHT, or FULL.
  • If you don’t use the keywords LEFT, RIGHT, or FULL you get an inner join.
Impact of having null values
Missing Information

• Two common scenarios:
  • Missing value.
    E.g., we know a student has some email address, but we don’t know what it is.
  • Inapplicable attribute.
    E.g., the value of attribute spouse for an unmarried person.
Representing missing information

• One possibility: use a special value as a placeholder. E.g.,
  • If age unknown, use 0.
  • If StNum unknown, use 999999999.

• Implications?

• Better solution: use a value not in any domain. We call this a null value.

• Tuples in SQL relations can have NULL as a value for one or more components.
Checking for null values

• You can compare an attribute value to **NULL** with
  
  • **IS NULL**
  
  • **IS NOT NULL**

• **Example:**

```
SELECT * 
FROM Course 
WHERE breadth IS NULL;
```
In SQL we have 3 truth-values

- Because of **NULL**, we need three truth-values:
  - If one or both operands to a comparison is **NULL**, the comparison always evaluates to **UNKNOWN**.
  - Otherwise, comparisons evaluate to **TRUE** or **FALSE**.
Combining truth values

• We need to know how the three truth-values combine with AND, OR and NOT.
• Can think of it in terms of the truth table.
• Or can think in terms of numbers:
  • \textbf{TRUE} = 1, \textbf{FALSE} = 0, \textbf{UNKNOWN} = 0.5
  • \textbf{AND} is min, \textbf{OR} is max,
  • \textbf{NOT} \ x is \ (1-x), \ i.e., \ it \ “flips” \ the \ value
# The three-valued truth table

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A and B</th>
<th>A or B</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>TF or FT</td>
<td>F</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>TU or UT</td>
<td>U</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td>FU or UF</td>
<td>F</td>
<td></td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>
Thinking of the truth-values as numbers

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>as nums</th>
<th>A and B</th>
<th>min</th>
<th>A or B</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>1, 1</td>
<td>T</td>
<td>1</td>
<td>T</td>
<td>1</td>
</tr>
<tr>
<td>TF or FT</td>
<td>1, 0</td>
<td>F</td>
<td>0</td>
<td>T</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>0, 0</td>
<td>F</td>
<td>0</td>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>TU or UT</td>
<td>1, 0.5</td>
<td>U</td>
<td>0.5</td>
<td>T</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>FU or UF</td>
<td>0, 0.5</td>
<td>F</td>
<td>0</td>
<td>U</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>0.5, 0.5</td>
<td>U</td>
<td>0.5</td>
<td>U</td>
<td>0.5</td>
</tr>
</tbody>
</table>
Surprises from 3-valued logic

- Some laws you are used to still hold in three-valued logic. For example,
  - **AND** is commutative.
- But others don’t. For example,
  - The law of the excluded middle breaks: \((p \lor (\neg p))\) might not be **TRUE**!
  - \((0 \times x)\) might not be **0**.
Impact of null values on WHERE

• A tuple is in a query result iff the WHERE clause is **TRUE**.
• **UNKNOWN** is not good enough.
• “WHERE is picky.”
• Example: **where-null**
Impact of null values on DISTINCT

- Example: `select-distinct-null`
- This behaviour may vary across DBMSs.
Impact of null values on aggregation

• Summary: Aggregation ignores **NULL**.
  - **NULL** never contributes to a sum, average, or count, and
  - can never be the minimum or maximum of a column (unless every value is **NULL**).
• If there are no *non-NULL* values in a column, then the result of the aggregation is **NULL**.
  • Exception: **COUNT** of an empty set is 0.
### Aggregation ignores nulls

<table>
<thead>
<tr>
<th></th>
<th>some nulls in A</th>
<th>All nulls in A</th>
</tr>
</thead>
<tbody>
<tr>
<td>min(A)</td>
<td>ignore the nulls</td>
<td>null</td>
</tr>
<tr>
<td>max(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sum(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>avg(A)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>count(A)</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>count(*)</td>
<td></td>
<td>all tuples count</td>
</tr>
</tbody>
</table>
Subqueries
Subqueries in a FROM clause

• In place of a relation name in the FROM clause, we can use a subquery.
• The subquery must be parenthesized.
• Must name the result, so you can refer to it in the outer query.
Example

• What does this do?

```sql
SELECT sid, dept||cnum as course, grade
FROM Took,
    (SELECT *
     FROM Offering
     WHERE instructor='Horton') Hoffering
WHERE Took.oid = Hoffering.oid;
```

• This FROM is analogous to:

```
Took × ρ_Hoffering ("subquery")
```

• Can you suggest another version?
Subquery as a value in a WHERE

• If a subquery is guaranteed to produce exactly one tuple, then the subquery can be used as a value.

• Simplest situation: that one tuple has only one component.
Example

• Find all students with a cgpa greater than that of student 99999.

SELECT sid, surname
FROM Student
WHERE cgpa >
  (SELECT cgpa
   FROM Student
   WHERE sid = 99999);

• This is analogous to something we can’t do in RA:

\[ \Pi_{\text{sid, surname}} \sigma_{\text{cgpa} > (\text{subquery})} \]
Special cases

• What if the subquery returns `NULL`?
• What if the subquery could return more than one value?
Quantifying over multiple results

• When a subquery can return multiple values, we can make comparisons using a quantifier.

• Previous example:

  ```sql
  SELECT sid, surname
  FROM Student
  WHERE cgpa >
    (SELECT cgpa
     FROM Student
     WHERE sid = 99999);
  ```

• We can require that gpa > all of them, or cgpa > at least one of them.
The Operator ANY

• Syntax:
  \[ x \ «comparison» ANY («subquery») \]
  or equivalently
  \[ x \ «comparison» SOME («subquery») \]

• Semantics:
  Its value is true iff the comparison holds for at least one tuple in the subquery result, i.e.,
  \[ \exists y \in «subquery results» \mid x \ «comparison» y \]

• \( x \) can be a list of attributes,
  but this feature is not supported by psql.
The Operator ALL

• Syntax:
  \( x \text{ «comparison» ALL («subquery»)} \)

• Semantics:
  Its value is true iff the comparison holds for every tuple in the subquery result, i.e.,
  \( \forall y \in \text{«subquery results»} \mid x \text{ «comparison» } y \)

• \( x \) can be a list of attributes, but this feature is not supported by psql.

• **Example:** any and all
The Operator IN

• Syntax:
  \[ x \text{ IN } (\text{«subquery»}) \]

• Semantics:
  Its value is true iff \( x \) equals at least one of the tuples in the subquery result.

• \( x \) can be a list of attributes, and psql does support this feature.
Example

What does this do?

SELECT sid, dept||cnum AS course, grade
FROM Took NATURAL JOIN Offering
WHERE
  grade >= 80 AND
  (cnum, dept) IN (
    SELECT cnum, dept
    FROM Took NATURAL JOIN Offering
    NATURAL JOIN Student
    WHERE surname = 'Lakemeyer');
Exercise

Suppose we have tables $R(a, b)$ and $S(b, c)$.

1. What does this query do?

   ```sql
   SELECT a
   FROM R
   WHERE b IN (SELECT b FROM S);
   ```

2. Can we express this query without using `IN`?
The Operator EXISTS

- Syntax:
  EXISTS («subquery»)

- Semantics:
  Its value is true iff the subquery has at least one tuple.

- Read it as “exists a row in the subquery result”
Example: EXISTS

```
SELECT surname, cgpa
FROM Student
WHERE EXISTS (  
    SELECT *  
    FROM Took  
    WHERE Student.sid = Took.sid and grade > 85 );
```
Scope

• Queries are evaluated from the inside out.
• If a name might refer to more than one thing, use the most closely nested one.
• If a subquery refers only to names defined inside it, it can be evaluated once and used repeatedly in the outer query.
• If it refers to any name defined outside of itself, it must be evaluated once for each tuple in the outer query.

These are called correlated subqueries.
Renaming can make scope explicit

```
SELECT instructor
FROM Offering Off1
WHERE NOT EXISTS (  
    SELECT *
    FROM Offering Off2
    WHERE  
        Off2.oid <> Off1.oid AND
        Off2.instructor = Off1.instructor
);  
```
Summary: where subqueries can go

- As a relation in a FROM clause.
- As a value in a WHERE clause.
- With ANY, ALL, IN or EXISTS in a WHERE clause.
- As operands to UNION, INTERSECT or EXCEPT.
- Reference: textbook, section 6.3.
Modifying a Database
Database Modifications

• Queries return a relation.
• A modification command does not; it changes the database in some way.
• Three kinds of modifications:
  • Insert a tuple or tuples.
  • Delete a tuple or tuples.
  • Update the value(s) of an existing tuple or tuples.
Two ways to insert

- We’ve already seen two ways to insert tuples into an empty table:

  INSERT INTO «relation» VALUES «list of tuples»;
  INSERT INTO «relation» («subquery») ;

- These can also be used to add tuples to a non-empty table.
Naming attributes in INSERT

• Sometimes we want to insert tuples, but we don’t have values for all attributes.
• If we name the attributes we are providing values for, the system will use NULL or a default for the rest.
• Convenient!
Example

CREATE TABLE Invite (  
    name TEXT,
    campus TEXT DEFAULT 'StG',
    email TEXT,
    age INT);

INSERT INTO Invite(name, email)  
(  SELECT firstname, email  
    FROM Student  
    WHERE cgpa > 3.4  )

Here, name and email get values from the query, campus gets the default value, and age gets NULL.
Deletion

• Delete tuples satisfying a condition:

```sql
DELETE FROM «relation»
WHERE «condition»;
```

• Delete all tuples:

```sql
DELETE FROM «relation»;
```
Example 1: Delete Some Tuples

DELETE FROM Course
WHERE NOT EXISTS (
    SELECT *
    FROM Took JOIN Offering
    ON Took.oid = Offering.oid
    WHERE
        grade > 50 AND
        Offering.dept = Course.dept AND
        Offering.cnum = Course.cnum
);
Updates

• To change the value of certain attributes in certain tuples to given values:

  UPDATE «relation»
  SET «list of attribute assignments»
  WHERE «condition on tuples»;

Example: update one tuple

- Updating one tuple:
  ```sql
  UPDATE Student
  SET campus = 'UTM'
  WHERE sid = 999999;
  ```

- Updating several tuples:
  ```sql
  UPDATE Took
  SET grade = 50
  WHERE grade >= 47 and grade < 50;
  ```