BETWEEN

... WHERE value BETWEEN 1 AND 5 ...

... WHERE date BETWEEN '2013-11-01' AND '2013-11-05' ...

• Same as $x \geq y \text{ AND } x \leq z$

• Note that dates without a time default to midnight beginning of day
  – True always, not just when using BETWEEN
UNION, INTERSECT, EXCEPT

query1 UNION query2 UNION query3 ...

• UNION includes the records returned from all the queries
  – The queries must be "union compatible"
    • Same number of columns
    • Corresponding columns have the same type
  – Eliminates duplicate rows (like DISTINCT) unless UNION ALL is used
UNION, INTERSECT, EXCEPT

query1 INTERSECT query2 INTERSECT query3 ...

• INTERSECT returns the rows that are in all the queries
  – Duplicates are eliminated unless INTERSECT ALL is used

query1 EXCEPT query2 EXCEPT query3 ...

• EXCEPT returns all rows that are in the first query but not the second
  – Duplicates are eliminated unless EXCEPT ALL is used
LIMIT and OFFSET

SELECT * FROM ... ORDER BY ... LIMIT number OFFSET number

- LIMIT limits the number of rows returns to the specified maximum
  - Might return less if the query matches less rows
- OFFSET skips the specified rows before beginning to return rows
- LIMIT and OFFSET only really make sense when used with ORDER BY to force a consistent ordering
Conditional Expressions

SELECT * FROM test;

| a |
|-|--|
| 1 |
| 2 |
| 3 |

SELECT
    CASE WHEN a=1 THEN 'one'
    WHEN a=2 THEN 'two'
    ELSE 'other'
END
FROM test;

case
------
one
two
other

• CASE is similar to if / then / else statement
• The result values must all have the same type
Conditional Expressions

• Simplified syntax when each case is a value
  – Similar to switch statement

```
SELECT
  CASE a
    WHEN 1 THEN 'one'
    WHEN 2 THEN 'two'
    ELSE 'other'
  END
FROM test;
```
Conditional Expressions

SELECT COALESCE (value1, value2, ...)

• COALESCE returns the first not null value
Window Function Calls

• A window function call applies an aggregate function to a portion of the rows selected by a query
  – A regular aggregate function call applies the function to the entire set of rows
  – Regular aggregate function calls require a GROUP BY to condense the output to a single row
  – Window function calls do not require a GROUP BY
Window Function Calls

• Regular aggregate function call

```
SELECT depname, avg(salary) FROM empsalary GROUP BY depname;
```

<table>
<thead>
<tr>
<th>depname</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>develop</td>
<td>5020.00000000000000000000000000000000</td>
</tr>
<tr>
<td>personnel</td>
<td>3700.00000000000000000000000000000000</td>
</tr>
<tr>
<td>sales</td>
<td>4866.6666666666666666666666666667</td>
</tr>
</tbody>
</table>

(3 rows)
Window Function Calls

- Window function call

```sql
SELECT depname, empno, salary, avg(salary)
    OVER (PARTITION BY depname) FROM empsalary;
```

<table>
<thead>
<tr>
<th>depname</th>
<th>empno</th>
<th>salary</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>develop</td>
<td>11</td>
<td>5200</td>
<td>5020.000000000000000000000000</td>
</tr>
<tr>
<td>develop</td>
<td>7</td>
<td>4200</td>
<td>5020.000000000000000000000000</td>
</tr>
<tr>
<td>develop</td>
<td>9</td>
<td>4500</td>
<td>5020.000000000000000000000000</td>
</tr>
<tr>
<td>develop</td>
<td>8</td>
<td>6000</td>
<td>5020.000000000000000000000000</td>
</tr>
<tr>
<td>develop</td>
<td>10</td>
<td>5200</td>
<td>5020.000000000000000000000000</td>
</tr>
<tr>
<td>personnel</td>
<td>5</td>
<td>3500</td>
<td>3700.000000000000000000000000</td>
</tr>
<tr>
<td>personnel</td>
<td>2</td>
<td>3900</td>
<td>3700.000000000000000000000000</td>
</tr>
<tr>
<td>sales</td>
<td>3</td>
<td>4800</td>
<td>4866.666666666666666666666667</td>
</tr>
<tr>
<td>sales</td>
<td>1</td>
<td>5000</td>
<td>4866.66666666666666666666666667</td>
</tr>
<tr>
<td>sales</td>
<td>4</td>
<td>4800</td>
<td>4866.66666666666666666666666667</td>
</tr>
</tbody>
</table>

(10 rows)
Window Function Calls

• Including an OVER clause after an aggregate function makes it a window function call
  – The PARTITION expression should split up the rows into various sets
    • The rows are partitioned after applying the WHERE, GROUP BY, and HAVING clauses
    • It is legal to not have a PARTITION expression, in which case the entire set of rows is a single partition
  – Every row is output with the aggregate functions computed over each row's *window frame*
Window Function Calls

• Window frames
  – Be default (without an ORDER BY clause) a row's window frame is the set of rows in the same partition as that row
  – If an ORDER BY expression is included in the OVER clause then the window frame of a row consists of all rows prior to and equal to that row in the partition
Window Function Calls

• Example: no ORDER BY
  – Calculates the sum over all of the rows

```
SELECT salary, sum(salary) OVER () FROM empsalary;
```

<table>
<thead>
<tr>
<th>salary</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>5200</td>
<td>47100</td>
</tr>
<tr>
<td>5000</td>
<td>47100</td>
</tr>
<tr>
<td>3500</td>
<td>47100</td>
</tr>
<tr>
<td>4800</td>
<td>47100</td>
</tr>
<tr>
<td>3900</td>
<td>47100</td>
</tr>
<tr>
<td>4200</td>
<td>47100</td>
</tr>
<tr>
<td>4500</td>
<td>47100</td>
</tr>
<tr>
<td>4800</td>
<td>47100</td>
</tr>
<tr>
<td>6000</td>
<td>47100</td>
</tr>
<tr>
<td>5200</td>
<td>47100</td>
</tr>
</tbody>
</table>

(10 rows)
Window Function Calls

• Example: ORDER BY
  – Calculates the sum up to and including that row and any equal rows

```
SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary;
```

<table>
<thead>
<tr>
<th>salary</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>3500</td>
<td>3500</td>
</tr>
<tr>
<td>3900</td>
<td>7400</td>
</tr>
<tr>
<td>4200</td>
<td>11600</td>
</tr>
<tr>
<td>4500</td>
<td>16100</td>
</tr>
<tr>
<td>4800</td>
<td>25700</td>
</tr>
<tr>
<td>4800</td>
<td>25700</td>
</tr>
<tr>
<td>5000</td>
<td>30700</td>
</tr>
<tr>
<td>5200</td>
<td>41100</td>
</tr>
<tr>
<td>5200</td>
<td>41100</td>
</tr>
<tr>
<td>5200</td>
<td>41100</td>
</tr>
<tr>
<td>6000</td>
<td>47100</td>
</tr>
</tbody>
</table>

(10 rows)
Window Function Calls

- Multiple aggregate function can be specified
  - Easier to write with a separate WINDOW clause (if the partitioning is the same)
  - Can actually have different partitions for different function calls

```
SELECT sum(salary) OVER w, avg(salary) OVER w
FROM empsalary
WINDOW w AS (PARTITION BY depname ORDER BY salary DESC);
```
WITH

• WITH lets you define subqueries to use in larger queries
  – The subqueries are called Common Table Expressions (CTEs)

WITH regional_sales AS (  
    SELECT region, SUM(amount) AS total_sales  
    FROM orders GROUP BY region
  ), top_regions AS (  
    SELECT region FROM regional_sales  
    WHERE total_sales >  
        (SELECT SUM(total_sales)/10 FROM regional_sales)
  )  
SELECT region, product, SUM(quantity) AS product_units,  
    SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
Question

• How can we represent a tree structure?
Trees

• Can be viewed as a many-to-one relationship from child to parent in the same table

• Option 1
  – "Foreign" key to parent node

```
CREATE TABLE nodes (  
id BIGSERIAL PRIMARY KEY,  
name TEXT NOT NULL  
parent_id BIGINT REFERENCES nodes(id)  
)
```
Trees

• Option 2
  – Mapping table

    CREATE TABLE nodes (  
      id BIGSERIAL PRIMARY KEY,  
      name TEXT NOT NULL  
    )

    CREATE TABLE node_map (  
      parent_id BIGINT REFERENCES nodes(id),  
      child_id BIGINT REFERENCES nodes(id),  
      PRIMARY KEY (parent_id, child_id)  
    )

• Other options exist
  – What if the sibling order matters?
Trees

• How do we get the immediate children of a node?

```
SELECT * FROM nodes WHERE parent_id = 5;
```

```
SELECT n.* FROM nodes n
  INNER JOIN node_map m ON n.id = m.child_id
WHERE m.parent_id = 5;
```

• How do we get all descendants of a node?
WITH RECURSIVE

A WITH RECURSIVE query can refer to its own output

WITH RECURSIVE t(n) AS (  
    VALUES (1)  
    UNION ALL  
    SELECT n+1 FROM t WHERE n < 5  
  )  
SELECT n FROM t;

n  
----  
1  
2  
3  
4  
5  
(5 rows)
WITH RECURSIVE

• General form

WITH RECURSIVE declaration AS ( 
    non-recursive term
    UNION [ALL]
    recursive term
)

query;
WITH RECURSIVE

• Process
  1. Evaluate the non-recursive term
  2. If UNION but not UNION ALL then eliminate duplicate rows
  3. Include remaining rows in output and place in a temporary working table
  4. While the working table is not empty
     1. Evaluate the recursive term substituting the current contents of the working table as the recursive term
     2. Eliminate duplicate rows and rows that duplicate previous results if not UNION ALL
     3. Include the remaining rows in the result and replace the working table with these rows
WITH RECURSIVE

- Often used to deal with hierarchical data (i.e., trees)

```
CREATE TABLE part_list (
    part TEXT,
    subpart TEXT,
    quantity INTEGER
);

WITH RECURSIVE included_parts(part, subpart, quantity) AS (
    SELECT root.part, root.subpart, root.quantity
    FROM part_list root
    WHERE root.part = '01'
    UNION ALL
    SELECT child.part, child.subpart, child.quantity
    FROM included_parts parent, part_list child
    WHERE parent.subpart = child.part
)
SELECT DISTINCT part, subpart, quantity
FROM included_parts
ORDER BY part, subpart, quantity;
```
WITH RECURSIVE

• Need to watch out for cycles!
  – If cycles are a possibility then it needs to be handled by the recursive query somehow

• Most RDBMSs do not require the RECURSIVE keyword
  – Though PostgreSQL does

• Oracle provides a different CONNECT BY construct for use with hierarchical data
  – PostgreSQL has a version of CONNECT BY now also
Pivoting

• Assume we have a table that looks like this

<table>
<thead>
<tr>
<th>student_id</th>
<th>assignment</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>Homework 1</td>
<td>A</td>
</tr>
<tr>
<td>12345</td>
<td>Homework 2</td>
<td>B</td>
</tr>
<tr>
<td>12345</td>
<td>Homework 3</td>
<td>A</td>
</tr>
<tr>
<td>23456</td>
<td>Homework 1</td>
<td>B</td>
</tr>
<tr>
<td>23456</td>
<td>Homework 2</td>
<td>A</td>
</tr>
<tr>
<td>23456</td>
<td>Homework 3</td>
<td>C</td>
</tr>
<tr>
<td>34567</td>
<td>Homework 1</td>
<td>C</td>
</tr>
<tr>
<td>34567</td>
<td>Homework 2</td>
<td>B</td>
</tr>
<tr>
<td>34567</td>
<td>Homework 3</td>
<td>A</td>
</tr>
</tbody>
</table>
Pivoting

• But we want output that looks like this

<table>
<thead>
<tr>
<th>student_id</th>
<th>Homework 1</th>
<th>Homework 2</th>
<th>Homework 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>23456</td>
<td>B</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>34567</td>
<td>C</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

• This is called a **pivot**
  – We are pivoting the rows to columns
Pivoting

- PostgreSQL has `crosstab` functions
  - Other RDBMSs use different syntax
- General form is

```sql
SELECT * FROM crosstab('source query', 'category query')
  AS (name type, name type, ...);
```

- The `source query` is a query to get the data
- The `category query` is a query to get the columns of the pivot
- A name and type must be provided for each column in the output
Pivoting

• **PostgreSQL has** `crosstab` **functions**
  – Other RDBMSs use different syntax

```sql
SELECT * FROM crosstab(
  'SELECT student_id, assignment, grade
   FROM grades order by student_id',
  'SELECT DISTINCT assignment
   FROM grades ORDER BY assignment'
) AS (student_id TEXT, assignment TEXT, grade TEXT);
```

• *(Probably easier, and more generic, to pivot in the application code)*