Advanced SQL and Functions

Stephen Frost
stephen.frost@crunchydatasolutions.com

Joe Conway
joe.conway@credativ.com

Crunchy Data Solutions, Inc.
credativ International

September 17, 2014
Queries
Syntax Overview

[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
* | expression [ [ AS ] output_name ] [, ...]
[ FROM from_item [, ...] ]
[ WHERE condition ]
[ GROUP BY expression, ... ]
[ HAVING condition [, ...] ]
[ WINDOW window_name AS ( window_definition ) [, ...] ]
[ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] .. ]
[ ORDER BY expression [ ASC | DESC | USING op ], ... ]
[ LIMIT num ] [ OFFSET num ]

Queries

Syntax Overview - from item

```
[ ONLY ] table_name [ * ]
  [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
[ LATERAL ] ( select )
  [ AS ] alias [ ( column_alias [, ...] ) ]
[ LATERAL ] function_name ( [ argument [, ...] ] )
  [ AS ] alias
    [ ( column_alias [, ...] | column_definition [, ...] ) ]
[ LATERAL ] function_name ( [ argument [, ...] ] )
  AS ( column_definition [, ...] )
with_query_name [ [ AS ] alias [ ( col_alias [, ...] ) ] ]
from_item [ NATURAL ] join_type
  from_item [ ON join_condition | USING ( column [, ...] ) ]

with_query_name [ ( column_name [, ...] ) ]
  AS ( select | values | insert | update | delete )
```
Last, but not least, the most complicated ones of all. VALUES returns a table, after evaluating all expressions:

```
VALUES ( expression [, ...] ) [, ...]
```

TABLE is essentially SELECT * FROM, but shorter:

```
TABLE table_name
    [ ORDER BY expression [ ASC | DESC | USING op ], ... ]
    [ LIMIT num ] [ OFFSET num ]
```
VALUES (1), (2), (3);

TABLE author;

SELECT * FROM author;

SELECT name AS designation FROM publisher;

SELECT DISTINCT authorid FROM bookauthor;

SELECT DISTINCT ON (authorid) authorid, bookname
FROM bookauthor ORDER BY authorid, bookpublishdate;

SELECT title FROM book
    WHERE net_price < 10.00;
SELECT * FROM book
    WHERE lower(title) LIKE '%sql%'
    ORDER BY lastname ASC, firstname ASC;

SELECT book.title FROM bookauthor, author, book
    WHERE author.lastname = 'Eisentraut'
        AND bookauthor.author_id = author.id
        AND book.id = bookauthor.book_id
    ORDER BY book.title DESC;

SELECT author, sum(price) AS price_sum
    FROM bookpricelist
        GROUP BY author HAVING sum(price) > 20
    ORDER BY author;
Join Types

- cross join
- inner join
- outer join
  - left
  - right
  - full
Cross Joins

Joins each row from the first table with each row from the second table

\[ \text{SELECT} \ \ast \ \text{FROM} \ \text{tab1} \ \text{CROSS JOIN} \ \text{tab2}; \]

is equivalent to

\[ \text{SELECT} \ \ast \ \text{FROM} \ \text{tab1, tab2}; \]

- Limited practical uses- generally used in error. particularly when comma-joins
- Use of comma-joins makes this more likely
- Useful for enumeration by creating a cartesian product
Inner Joins

Joins each row of the first table with each row from the second table for which the condition matches

SELECT ... FROM tab1 [ INNER ] JOIN tab2 ON condition;

SELECT ... FROM tab1 [ INNER ] JOIN tab2 USING (column list);

SELECT ... FROM tab1 NATURAL [ INNER ] JOIN tab2;

Or using the “traditional” (horrible) comma-join notation:

SELECT ... FROM tab1, tab2 WHERE condition;
Inner Joins
Examples

```sql
SELECT * FROM book INNER JOIN publisher
ON book.publisher_id = publisher.id;

SELECT * FROM bibo INNER JOIN author
USING (book_id);
```
Outer Joins

Joins each row from the first table with each row from the second table for which the condition matches. Furthermore, nonmatching rows are added to the result.

- **left join** all rows from the left table
- **right join** all rows from the right table
- **full join** all rows from both tables

Rows without a join partner are filled up with null values.
Outer Joins

Syntax

```sql
SELECT ... FROM tab1 LEFT/RIGHT/FULL [ OUTER ] JOIN tab2
   ON condition;
```

```sql
SELECT ... FROM tab1 LEFT/RIGHT/FULL [ OUTER ] JOIN tab2
   USING (column list);
```

```sql
SELECT ... FROM tab1 NATURAL LEFT/RIGHT/FULL [ OUTER ] JOIN tab2;
```
Outer Joins

Examples

```sql
SELECT * FROM book RIGHT JOIN bookcategory
    ON book.id = bookcategory.category_id;

SELECT * FROM publisher LEFT JOIN book
    ON publisher.id = book.publisher_id;
```
### Example Data

```sql
sfrost=# table book;

<table>
<thead>
<tr>
<th>title</th>
<th>language</th>
<th>price</th>
<th>authorname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Free</td>
<td>English</td>
<td>100.00</td>
<td>Stephen Frost</td>
</tr>
<tr>
<td>Running Wild</td>
<td>English</td>
<td>80.00</td>
<td>Stephen Frost</td>
</tr>
<tr>
<td>Running Scared</td>
<td>English</td>
<td>50.00</td>
<td>Stephen Frost</td>
</tr>
<tr>
<td>Kostenlos Laufen</td>
<td>German</td>
<td>95.00</td>
<td>Joe Conway</td>
</tr>
<tr>
<td>Wildlauf</td>
<td>German</td>
<td>75.00</td>
<td>Joe Conway</td>
</tr>
<tr>
<td>Angst Lauf</td>
<td>German</td>
<td>45.00</td>
<td>Joe Conway</td>
</tr>
</tbody>
</table>

(6 rows)
```
SELECT title FROM book
UNION
SELECT authorname FROM book;

<table>
<thead>
<tr>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>Joe Conway</td>
</tr>
<tr>
<td>Kostenlos Laufen</td>
</tr>
<tr>
<td>Angst Lauf</td>
</tr>
<tr>
<td>Wildlauf</td>
</tr>
<tr>
<td>Running Free</td>
</tr>
<tr>
<td>Running Scared</td>
</tr>
<tr>
<td>Running Wild</td>
</tr>
<tr>
<td>Stephen Frost</td>
</tr>
</tbody>
</table>

(8 rows)
Set Operations
UNION ALL

SELECT title FROM book
UNION ALL
SELECT authorname FROM book;

title
------------------
Running Free
Running Wild
Running Scared
Kostenlos Laufen
Wildlauf
Angst Lauf
Stephen Frost
Stephen Frost
Stephen Frost
Joe Conway
Joe Conway
Joe Conway
(12 rows)
Set Operations
INTERSECT

SELECT title FROM book
INTERSECT
SELECT authorname FROM book;

title
-------
(0 rows)
Set Operations
EXCEPT

SELECT title FROM book
EXCEPT
SELECT authorname FROM book;

<table>
<thead>
<tr>
<th>title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Free</td>
</tr>
<tr>
<td>Running Scared</td>
</tr>
<tr>
<td>Wildlauf</td>
</tr>
<tr>
<td>Running Wild</td>
</tr>
<tr>
<td>Angst Lauf</td>
</tr>
<tr>
<td>Kostenlos Laufen</td>
</tr>
</tbody>
</table>

(6 rows)
Subqueries

Uncorrelated

Uncorrelated subquery:

- Subquery calculates a constant result set for the upper query
- Executed only once

```
SELECT title, authorname, price
FROM book
WHERE book.price >
    (SELECT AVG(book.price) FROM book);
```

<table>
<thead>
<tr>
<th>title</th>
<th>authorname</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Free</td>
<td>Stephen Frost</td>
<td>100.00</td>
</tr>
<tr>
<td>Running Wild</td>
<td>Stephen Frost</td>
<td>80.00</td>
</tr>
<tr>
<td>Kostenlos Laufen</td>
<td>Joe Conway</td>
<td>95.00</td>
</tr>
<tr>
<td>Wildlauf</td>
<td>Joe Conway</td>
<td>75.00</td>
</tr>
</tbody>
</table>

(4 rows)
Correlated subquery:

- Subquery references variables from the upper query
- Subquery has to be repeated for each row of the upper query
- Could be rewritten as a join

```sql
SELECT title, authorname, price
FROM book book_outer
WHERE EXISTS
    (SELECT * FROM book WHERE
     book_outer.language = book.language
    AND book.id <> book_outer.id
    AND abs(book_outer.price - book.price) <= 20);
```
### Subqueries

**Correlated**

Results:

<table>
<thead>
<tr>
<th>title</th>
<th>authorname</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Free</td>
<td>Stephen Frost</td>
<td>100.00</td>
</tr>
<tr>
<td>Running Wild</td>
<td>Stephen Frost</td>
<td>80.00</td>
</tr>
<tr>
<td>Kostenlos Laufen</td>
<td>Joe Conway</td>
<td>95.00</td>
</tr>
<tr>
<td>Wildlauf</td>
<td>Joe Conway</td>
<td>75.00</td>
</tr>
</tbody>
</table>

(4 rows)
Window Functions - Basics

Window functions are like ordinary aggregates, but are restricted to operate on a portion of the tuples only.

```
function_name ([expression [, expression ... ]]) OVER ( window_definition )
function_name ([expression [, expression ... ]]) OVER window_name
function_name ( * ) OVER ( window_definition )
function_name ( * ) OVER window_name
```

Where `window_name` is an identifier and `window_definition` is:

```
[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]
```

Each Window function scans all tuples belonging to the “group” the current tuple is part of.
Window Functions - Frame Clause

Range vs. Rows

- **RANGE UNBOUNDED PRECEDING**
  select all rows from the partition start up through the last peer in
  the order of its ORDER BY clause (or all if omitted)

- **RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING**
  select all rows in the partition

- **ROWS UNBOUNDED PRECEDING**
  select all rows (regardless of duplicates) up through the current row
  in order of its ORDER BY clause

- **BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**
  same as UNBOUNDED PRECEDING
Window Functions Basic Example

Select all books and compare its price against the average price of all books in the same language:

```
SELECT title, language, price, AVG(price) OVER(PARTITION BY language) FROM book;
```

<table>
<thead>
<tr>
<th>title</th>
<th>language</th>
<th>price</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Free</td>
<td>English</td>
<td>100.00</td>
<td>76.6666666666666667</td>
</tr>
<tr>
<td>Running Wild</td>
<td>English</td>
<td>80.00</td>
<td>76.6666666666666667</td>
</tr>
<tr>
<td>Running Scared</td>
<td>English</td>
<td>50.00</td>
<td>76.6666666666666667</td>
</tr>
<tr>
<td>Kostenlos Laufen</td>
<td>German</td>
<td>95.00</td>
<td>71.6666666666666667</td>
</tr>
<tr>
<td>Wildlauf</td>
<td>German</td>
<td>75.00</td>
<td>71.6666666666666667</td>
</tr>
<tr>
<td>Angst Lauf</td>
<td>German</td>
<td>45.00</td>
<td>71.6666666666666667</td>
</tr>
</tbody>
</table>

(6 rows)
Window Function results can be operated against similar to functions-

```
SELECT title, language, price,
       ROUND(AVG(price) OVER(PARTITION BY language),2) FROM book;
```

```
title      | language | price  | round  
----------------+----------+--------+-------
Running Free    | English  | 100.00 | 76.67 |
Running Wild    | English  |  80.00 | 76.67 |
Running Scared  | English  |  50.00 | 76.67 |
Kostenlos Laufen| German   |  95.00 | 71.67 |
Wildlauf        | German   |  75.00 | 71.67 |
Angst Lauf      | German   |  45.00 | 71.67 |
```

(6 rows)
Window Functions Range vs Row Example

With RANGE, all duplicates are considered part of the same group and the function is run across all of them, with the same result used for all members of the group.

```
SELECT title, language, price,
    AVG(price) OVER(ORDER BY language RANGE UNBOUNDED PRECEDING)
FROM book;
```

<table>
<thead>
<tr>
<th>title</th>
<th>language</th>
<th>price</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Free</td>
<td>English</td>
<td>100.00</td>
<td>76.6666666666666667</td>
</tr>
<tr>
<td>Running Wild</td>
<td>English</td>
<td>80.00</td>
<td>76.6666666666666667</td>
</tr>
<tr>
<td>Running Scared</td>
<td>English</td>
<td>50.00</td>
<td>76.6666666666666667</td>
</tr>
<tr>
<td>Kostenlos Laufen</td>
<td>German</td>
<td>95.00</td>
<td>74.1666666666666667</td>
</tr>
<tr>
<td>Wildlauf</td>
<td>German</td>
<td>75.00</td>
<td>74.1666666666666667</td>
</tr>
<tr>
<td>Angst Lauf</td>
<td>German</td>
<td>45.00</td>
<td>74.1666666666666667</td>
</tr>
</tbody>
</table>

(6 rows)
With rows, can get a "running" average even with an ORDER BY over duplicates:

```
SELECT
    title, language, price,
    AVG(price) OVER(ORDER BY language ROWS UNBOUNDED PRECEDING)
FROM book;
```

<table>
<thead>
<tr>
<th>title</th>
<th>language</th>
<th>price</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Free</td>
<td>English</td>
<td>100.00</td>
<td>100.000000000000000000000000000000000000</td>
</tr>
<tr>
<td>Running Wild</td>
<td>English</td>
<td>80.00</td>
<td>90.000000000000000000000000000000000000</td>
</tr>
<tr>
<td>Running Scared</td>
<td>English</td>
<td>50.00</td>
<td>76.666666666666666666666666666666666666</td>
</tr>
<tr>
<td>Kostenlos Laufen</td>
<td>German</td>
<td>95.00</td>
<td>81.250000000000000000000000000000000000</td>
</tr>
<tr>
<td>Wildlauf</td>
<td>German</td>
<td>75.00</td>
<td>80.000000000000000000000000000000000000</td>
</tr>
<tr>
<td>Angst Lauf</td>
<td>German</td>
<td>45.00</td>
<td>74.166666666666666666666666666666666666</td>
</tr>
</tbody>
</table>

(6 rows)
Window Functions Window Clause

Select all books and compare its price against the average price and total price of all books in the same language:

```
SELECT title, language, price,
       AVG(price) OVER mywindow,
       SUM(price) OVER mywindow
FROM book
WINDOW mywindow AS (PARTITION BY language);
```

Window Functions

Window Function - Examples

Results:

<table>
<thead>
<tr>
<th>title</th>
<th>language</th>
<th>price</th>
<th>avg</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Free</td>
<td>English</td>
<td>100.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
<tr>
<td>Running Wild</td>
<td>English</td>
<td>80.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
<tr>
<td>Running Scared</td>
<td>English</td>
<td>50.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
<tr>
<td>Kostenlos Laufen</td>
<td>German</td>
<td>95.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
<tr>
<td>Wildlauf</td>
<td>German</td>
<td>75.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
<tr>
<td>Angst Lauf</td>
<td>German</td>
<td>45.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
</tbody>
</table>

(6 rows)
Multiple Window clauses can be in the same query, or even some with a named window clause and some without one.

```sql
SELECT
  row_number() OVER () as row,
  title, language, price,
  AVG(price) OVER mywindow,
  SUM(price) OVER mywindow
FROM book
WINDOW mywindow AS (PARTITION BY language);
```
### Window Functions Row Number

Results:

<table>
<thead>
<tr>
<th>row</th>
<th>title</th>
<th>language</th>
<th>price</th>
<th>avg</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Running Free</td>
<td>English</td>
<td>100.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
<tr>
<td>2</td>
<td>Running Wild</td>
<td>English</td>
<td>80.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
<tr>
<td>3</td>
<td>Running Scared</td>
<td>English</td>
<td>50.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
<tr>
<td>4</td>
<td>Kostenlos Laufen</td>
<td>German</td>
<td>95.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
<tr>
<td>5</td>
<td>Wildlauf</td>
<td>German</td>
<td>75.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
<tr>
<td>6</td>
<td>Angst Lauf</td>
<td>German</td>
<td>45.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
</tbody>
</table>

(6 rows)
### Window Functions Rank

```sql
SELECT
    rank() OVER (ORDER BY title), title, language, price,
    AVG(price) OVER mywindow,
    SUM(price) OVER mywindow
FROM book
WINDOW mywindow AS (PARTITION BY language);
```

<table>
<thead>
<tr>
<th>rank</th>
<th>title</th>
<th>language</th>
<th>price</th>
<th>avg</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Angst Lauf</td>
<td>German</td>
<td>45.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
<tr>
<td>2</td>
<td>Kostenlos Laufen</td>
<td>German</td>
<td>95.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
<tr>
<td>3</td>
<td>Running Free</td>
<td>English</td>
<td>100.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
<tr>
<td>4</td>
<td>Running Scared</td>
<td>English</td>
<td>50.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
<tr>
<td>5</td>
<td>Running Wild</td>
<td>English</td>
<td>80.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
<tr>
<td>6</td>
<td>Wildlauf</td>
<td>German</td>
<td>75.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
</tbody>
</table>

(6 rows)
ORDER BY window clause may re-order the rows, but an explicit overall ORDER BY can still be used to achieve the desired result ordering.

```sql
SELECT
    rank() OVER (ORDER BY title), title, language, price,
    AVG(price) OVER mywindow,
    SUM(price) OVER mywindow
FROM book
WINDOW mywindow AS (PARTITION BY language) ORDER BY price;
```
Window Functions Rank

Note that the rank value remains correct even though the final ordering is changed.

Results:

<table>
<thead>
<tr>
<th>rank</th>
<th>title</th>
<th>language</th>
<th>price</th>
<th>avg</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Angst Lauf</td>
<td>German</td>
<td>45.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
<tr>
<td>4</td>
<td>Running Scared</td>
<td>English</td>
<td>50.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
<tr>
<td>6</td>
<td>Wildlauf</td>
<td>German</td>
<td>75.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
<tr>
<td>5</td>
<td>Running Wild</td>
<td>English</td>
<td>80.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
<tr>
<td>2</td>
<td>Kostenlos Laufen</td>
<td>German</td>
<td>95.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
<tr>
<td>3</td>
<td>Running Free</td>
<td>English</td>
<td>100.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
</tbody>
</table>

(6 rows)
Window Functions Rank Duplicates

Rank handles duplicates also. Note we are ranking over language now.

```
SELECT
    rank() OVER (ORDER BY language), title, language, price,
    AVG(price) OVER mywindow,
    SUM(price) OVER mywindow
FROM book
WINDOW mywindow AS (PARTITION BY language);
```

<table>
<thead>
<tr>
<th>rank</th>
<th>title</th>
<th>language</th>
<th>price</th>
<th>avg</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Running Free</td>
<td>English</td>
<td>100.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
<tr>
<td>1</td>
<td>Running Wild</td>
<td>English</td>
<td>80.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
<tr>
<td>1</td>
<td>Running Scared</td>
<td>English</td>
<td>50.00</td>
<td>76.6666666666666667</td>
<td>230.00</td>
</tr>
<tr>
<td>4</td>
<td>Kostenlos Laufen</td>
<td>German</td>
<td>95.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
<tr>
<td>4</td>
<td>Wildlauf</td>
<td>German</td>
<td>75.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
<tr>
<td>4</td>
<td>Angst Lauf</td>
<td>German</td>
<td>45.00</td>
<td>71.6666666666666667</td>
<td>215.00</td>
</tr>
</tbody>
</table>

(6 rows)
Introduction - What is a CTE?

- SQL Standard defines table expressions declared by **WITH**
- A table expression is noted as **Common Table Expression, CTE**
- A CTE could contain recursive references to itself
- Can be seen as a temp table or view private to a query
Syntax

```
WITH [ RECURSIVE ] with_query [, ...] ]

with with_query as:

with_query_name [ ( column_name [, ...] ) ]
    AS ( select )

Using a self-reference within a RECURSIVE query needs the following syntax in the inner WITH definition:

non_recursive_term UNION [ ALL ] recursive_term
```
Simple non-recursive example

```sql
SELECT (SELECT 'Stephen'::text) || ' ' || (SELECT 'Frost'::text);

WITH as syntactic sugar:

WITH with_1(prenames)
AS ( SELECT 'Stephen'::text ),
with_2(fullname)
AS ( SELECT with_1.prenames || ' ' || 'Frost' FROM with_1 )
SELECT fullname FROM with_2;

Result: Stephen Frost
```
Another non-recursive example

Use WITH clauses to calculate the average by language, then another to pull the sum by language, and finally join them with the original table.

```
WITH avg_price(language, avg) AS ( SELECT language, avg(price) FROM book GROUP BY language ),
sum_price(language, sum) AS ( SELECT language, sum(price) FROM book GROUP BY language )
SELECT book.*, round(avg,2) as avg, round(sum,2) as sum FROM book
JOIN avg_price USING (language)
JOIN sum_price USING (language);
```
### Another non-recursive example

Results:

<table>
<thead>
<tr>
<th>title</th>
<th>language</th>
<th>price</th>
<th>authorname</th>
<th>id</th>
<th>avg</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Free</td>
<td>English</td>
<td>100.00</td>
<td>Stephen Frost</td>
<td>1</td>
<td>76.67</td>
<td>230.00</td>
</tr>
<tr>
<td>Running Wild</td>
<td>English</td>
<td>80.00</td>
<td>Stephen Frost</td>
<td>2</td>
<td>76.67</td>
<td>230.00</td>
</tr>
<tr>
<td>Running Scared</td>
<td>English</td>
<td>50.00</td>
<td>Stephen Frost</td>
<td>3</td>
<td>76.67</td>
<td>230.00</td>
</tr>
<tr>
<td>Kostenlos Laufen</td>
<td>German</td>
<td>95.00</td>
<td>Joe Conway</td>
<td>4</td>
<td>71.67</td>
<td>215.00</td>
</tr>
<tr>
<td>Wildlauf</td>
<td>German</td>
<td>75.00</td>
<td>Joe Conway</td>
<td>5</td>
<td>71.67</td>
<td>215.00</td>
</tr>
<tr>
<td>Angst Lauf</td>
<td>German</td>
<td>45.00</td>
<td>Joe Conway</td>
<td>6</td>
<td>71.67</td>
<td>215.00</td>
</tr>
</tbody>
</table>

(6 rows)
Simple recursive example

List all numbers from 1 to 100:

WITH RECURSIVE foo_with(n)
AS
(  VALUES(1)
      UNION
      SELECT  
          n+1
      FROM     
          foo_with
      WHERE n < 100
  ) SELECT * FROM foo_with ORDER BY n;
Recursion - formal explanation

*IT* is the Intermediate Table, *WT* the Working Table and *RT* the Result Table.

1. Initialize
   - *IT* is initialized as an empty set
   - Execute the non-recursive query
   - Assign results to both *RT* and *WT*;

2. Execute recursive query
   - Replace recursive self-reference with *WT*
   - Assign results during execution to *IT*
   - Append *IT* to *RT*
   - Replace *WT* with current *IT*
   - Truncate *IT*

3. Check recursion
   - Repeat 2) until *WT* is an empty set
   - Return *RT*
Simple recursive example

Detailed example:

WITH RECURSIVE foo_with(n)
AS (  

    -- non-recursive query, assign results to WT, RT
    VALUES(1)

    -- recursive query with self reference to foo_with
    -- self-reference substituted by WT, results
    -- assigned to IT, WT and appended to RT

    UNION
    SELECT
        n+1
    FROM
        foo_with
    WHERE n < 100

    -- empty IT and execute recursive term as long
    -- as WT contains any tuple.

)

-- produces result set RT

SELECT * FROM foo_with ORDER BY n;
Another example (1)

CREATE TABLE parts_list
  (whole text, part text, count int);

INSERT INTO parts_list VALUES
  ('car', 'engine', 1),
  ('car', 'wheel', 4),
  ('engine', 'cylinder head', 1),
  ('cylinder head', 'screw', 14),
  ('wheel', 'screw', 5),
  ('car', 'doors', 4),
  ('car', 'steering wheel', 1),
  ('doors', 'window', 1);

*parts_list* is a self-referencing table, cannot be easily retrieved with plain SQL.
Another example (2)

Return the number of screws needed to assemble a car

WITH RECURSIVE list(whole, part, num)
AS
(
    SELECT whole, part, count AS num
    FROM parts_list
    WHERE whole = 'car'
    UNION

    SELECT d.whole, d.part, d.count * list.num AS num
    FROM list
    JOIN parts_list d ON (d.whole = list.part)
)
SELECT SUM(num) FROM list WHERE part = 'screw';

Result: 34
 Recursive queries use iteration in reality

**UNION** vs. **UNION ALL**

- Only one recursive self-reference allowed
- Primary query evaluates subqueries defined by **WITH** only once
- Name of a **WITH**-Query hides any “real” table
- No aggregates, **GROUP BY**, **HAVING**, **ORDER BY**, **LIMIT**, **OFFSET** in a recursive query allowed
- No mutual recursive **WITH**-Queries allowed
- Recursive references must not be part of an OUTER JOIN

Simple Writable CTE

Delete from one table and insert into another

WITH archive_rows() AS

(DELETE
  FROM parts_list
  WHERE whole = 'car'
  RETURNING *
)

INSERT INTO parts_list_archive
  SELECT * FROM archive_rows;
Recursive Writable CTE

Insert the parts needed to assemble a car into another table

WITH RECURSIVE list(whole, part, num)
AS
(
    SELECT whole, part, count AS num
    FROM parts_list
    WHERE whole = 'car'

    UNION

    SELECT d.whole, d.part, d.count * list.num AS num
    FROM list
    JOIN parts_list d ON (d.whole = list.part)
)

INSERT INTO parts_list_car SELECT * FROM list;
LATERAL is a new JOIN method (aka ’LATERAL JOIN’) which allows a subquery in one part of the FROM clause to reference columns from earlier items in the FROM clause.

- Refer to earlier table
- Refer to earlier subquery
- Refer to earlier set-returning function

Implicitly added when a SRF is referring to an earlier item in the FROM clause.
LATERAL Table example

Refer to earlier table's column in arguments to SRF

CREATE TABLE numbers AS
    SELECT generate_series as max_num FROM generate_series(1,10);

SELECT *
FROM numbers, LATERAL generate_series(1,max_num);

SELECT *
FROM numbers, generate_series(1,max_num);
LATERAL Table example

```sql
SELECT *
FROM numbers, generate_series(1,max_num);
```

```
<table>
<thead>
<tr>
<th>max_num</th>
<th>generate_series</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
```

[...]  
(55 rows)
LATERAL Subquery example

Refer to earlier subquery’s column in arguments to SRF
Returns same results as previous query, but without the table.

```sql
SELECT *
FROM (SELECT generate_series as max_num
       FROM generate_series(1,10)) as numbers,
     LATERAL generate_series(1,max_num);
```

```sql
SELECT *
FROM (SELECT generate_series as max_num
       FROM generate_series(1,10)) as numbers,
       generate_series(1,max_num);
```
LATERAL Subquery example

Refer to earlier subquery’s column in arguments to SRF

```
SELECT *
FROM (SELECT generate_series as max_num
      FROM generate_series(1,10)) as numbers,
     LATERAL generate_series(1,max_num);
```

```
SELECT *
FROM (SELECT generate_series as max_num
      FROM generate_series(1,10)) as numbers,
     generate_series(1,max_num);
```

```
SELECT * FROM generate_series(1,10) as numbers,
     generate_series(1,numbers);
```
JSONB vs. JSON

JSONB is a new data type in 9.4 which is nearly identical to the JSON data type. There are a few specific difference which are important to note:

- JSON is stored as a regular 'text' blob, making it slow to utilize
- JSONB is stored much more efficiently in a binary data format
- JSONB is very slightly slower to input
- JSONB normalizes input, reduces whitespace, does not preserve order or duplicates
- JSON can only be sensibly indexed through functional indexes
- JSONB can be directly indexed
- JSONB number output depends on PostgreSQL numeric data type
- JSONB has containment and existence operators
As mentioned, JSONB does not preserve whitespace (or lack of it), for example:

```sql
postgres=# SELECT '{"bar":"baz","balance":7.77,"active":false}':::json;
 json
---------------------------------------------
 {"bar":"baz","balance":7.77,"active":false}
(1 row)
```

```sql
postgres=# SELECT '{"bar":"baz","balance":7.77,"active":false}':::jsonb;
 jsonb
--------------------------------------------------
 {"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```
JSONB Example

JSONB uses the numeric data type's output format, see these two identical inputs:

```
postgres=# SELECT '{"reading": 1.230e-5} '::json, '{"reading": 1.230e-5} '::jsonb
                          json |            jsonb
-------------------------------+-------------------------
{"reading": 1.230e-5} | {"reading": 0.00001230}
(1 row)
```
JSONB Containment and Existance Examples

Array on the right side is contained within the one on the left.

```
postgres=# SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;
 ?column?
----------
t
(1 row)
```

JSONB structure on the left contains the element on the right.

```
postgres=# SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';
 ?column?
----------
t
(1 row)
```
SP-GIST differs from other index types by decomposing the given space into disjoint partitions.

- SP-GIST index creation is generally faster than GIST
- SP-GIST index size is comparable to GIST
- SP-GIST query time is much faster than GIST
postgres=# create table geo (point point);
CREATE TABLE
postgres=# create index pt_gist_idx on geo using gist(point);
CREATE INDEX
postgres=# create index pt_spgist_idx on geo using spgist(point);
CREATE INDEX
postgres=# insert into geo
postgres=# select (random()*180-90 || ',,' || random()*360-180)::point
postgres=# from generate_series(1,1000000);
INSERT 0 1000000
Performance depends on the amount of data and the size of the overall space of the data which is indexed. A simple 1,000,000 point example shows improved performance, where smaller data sets showed little difference:

```
postgres=# explain analyze select point from geo where point ~= '(-29.5491208042949,15.2670755423605)';
[...]
Execution time: 0.245 ms
postgres=# create index pt_spgist_idx on geo using spgist(point);
CREATE INDEX
postgres=# explain analyze select point from geo where point ~= '(-29.5491208042949,15.2670755423605)';
[...]
Execution time: 0.158 ms
```
What are Functions?

- Full fledged SQL objects
- Many other database objects are implemented with them
- Fundamental part of PostgreSQL’s system architecture
- Created with CREATE FUNCTION
- Executed through normal SQL
  
  - target-list:
    SELECT myfunc(f1) FROM foo;
  
  - FROM clause:
    SELECT * FROM myfunc();
  
  - WHERE clause:
    SELECT * FROM foo WHERE myfunc(f1) = 42;
How are they Used?

- Functions
- Operators
- Data types
- Index methods
- Casts
- Triggers
- Aggregates
- Ordered-set Aggregates
- Window Functions
What Forms Can They Take?

- PostgreSQL provides four kinds of functions:
  - SQL
  - Procedural Languages
  - Internal
  - C-language
  
- Arguments
  - Base, composite, or combinations
  - Scalar or array
  - Pseudo or polymorphic
  - VARIADIC
  - IN/OUT/INOUT

- Return
  - Singleton or set (SETOF)
  - Base or composite type
  - Pseudo or polymorphic
SQL Functions

- **Behavior**
  - Executes an arbitrary list of SQL statements separated by semicolons
  - Last statement may be INSERT, UPDATE, or DELETE with RETURNING clause

- **Arguments**
  - Referenced by function body using name or $n: $1 is first arg, etc.
  - If composite type, then dot notation $1.name used to access
  - Only used as data values, not as identifiers

- **Return**
  - If singleton, first row of last query result returned, NULL on no result
  - If SETOF, all rows of last query result returned, empty set on no result
Procedural Languages

- User-defined functions
- Written in languages besides SQL and C
  - Task is passed to a special handler that knows the details of the language
  - Dynamically loaded
  - Could be self-contained (e.g. PL/pgSQL)
  - Might be externally linked (e.g. PL/Perl)

http://www.postgresql.org/docs/9.4/static/xplang.html
Internal Functions

- Statically linked C functions
  - Could use CREATE FUNCTION to create additional alias names for an internal function
  - Most internal functions expect to be declared STRICT

CREATE FUNCTION square_root(double precision)
RETURNS double precision AS
'dsqr\t'
LANGUAGE internal STRICT;

http://www.postgresql.org/docs/9.4/static/xfunc-internal.html
C Language Functions

- User-defined functions written in C
  - Compiled into dynamically loadable objects (also called shared libraries)
  - Loaded by the server on demand
  - contrib is good source of examples
  - Same as internal function coding conventions
  - Require PG_MODULE_MAGIC call
  - Short example later, but deserves separate tutorial

http://www.postgresql.org/docs/9.4/static/xfunc-c.html
PostgreSQL includes the following server-side procedural languages:

http://www.postgresql.org/docs/9.4/static/xplang.html

- PL/pgSQL
- Perl
- Python
- Tcl

Other languages available:


- Java
- V8 (Javascript)
- Ruby
- R
- Shell
- others ...
Creating New Functions

CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
    [ RETURNS rettype
    | RETURNS TABLE ( column_name column_type [, ...] ) ]
{ LANGUAGE lang_name
 | WINDOW
 | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
 | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
 | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
 | COST execution_cost
 | ROWS result_rows
 | SET configuration_parameter { TO value | = value | FROM CURRENT }
 | AS 'definition'
 | AS 'obj_file', 'link_symbol'
} ...
[ WITH ( attribute [, ...] ) ]
Dollar Quoting

- Works for all character strings
- Particularly useful for function bodies
- Consists of a dollar sign ($), "tag" of zero or more characters, another dollar sign
- Start and End tag must match
- Nest dollar-quoted string literals by choosing different tags at each nesting level

```
CREATE OR REPLACE FUNCTION dummy () RETURNS text AS
$_$
    BEGIN
        RETURN $$Say 'hello'$$;
    END;
$_$
LANGUAGE plpgsql;
```
DO [ LANGUAGE lang_name ] code

- Keyword DO executes anonymous code block
- Transient
- Any procedural language with support, defaults to plpgsql
- No parameters, returns void
- Parsed and executed once
- LANGUAGE clause can be before or after code block

http://www.postgresql.org/docs/9.4/static/sql-do.html
DO $$
DECLARE r record;
BEGIN
  FOR r IN SELECT u.rolname
    FROM pg_authid u
    JOIN pg_auth_members m on m.member = u.oid
    JOIN pg_authid g on g.oid = m.roleid
    WHERE g.rolname = 'admin'
  LOOP
    EXECUTE $$ ALTER ROLE $$ || r.rolname ||
    $$ SET work_mem = '512MB' $$;
  END LOOP;
END$$;
Anonymous Functions

```sql
SELECT u.rolname, s.setconfig as setting
FROM pg_db_role_setting s
JOIN pg_authid u on u.oid = s.setrole
JOIN pg_auth_members m on m.member = u.oid
JOIN pg_authid g on g.oid = m.roleid
WHERE g.rolname = 'admin';
```

<table>
<thead>
<tr>
<th>rolname</th>
<th>setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>rockstar</td>
<td>{work_mem=512MB}</td>
</tr>
</tbody>
</table>

(1 row)
Changing Existing Functions

- Once created, dependent objects may be created
- Must do DROP FUNCTION ... CASCADE to recreate
- Or use OR REPLACE to avoid dropping dependent objects
- Very useful for large dependency tree
- Can't be used in some circumstances (must drop/recreate instead). You cannot:
  - change function name or argument types
  - change return type
  - change types of any OUT parameters

CREATE OR REPLACE FUNCTION ...;
Function Arguments - argmode

```
( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
```

- **argmode** (optional): IN, OUT, INOUT, or VARIADIC
  - IN is the default if argmode is omitted
  - OUT and INOUT cannot be used with RETURNS TABLE
  - VARIADIC can only be followed by OUT
  - Not required (but good style): IN, then INOUT, then OUT
  - Func name + IN/INOUT/VARIADIC arg sig identifies function

CREATE FUNCTION testfoo (IN int, INOUT int, OUT int)
RETURNS RECORD AS $$
VALUES ($2, $1 * $2);
$$ language sql;
SELECT * FROM testfoo(14, 3);
```
column1 | column2
---------+---------
3 | 42
(1 row)
```
### Function Arguments - argname

```sql
( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
```

- **argname (optional):**
  - Most, but not all, languages will use in function body
  - Use named notation to improve readability and allow reordering
  - Defines the OUT column name in the result row type

```sql
CREATE FUNCTION testfoo (IN a int, INOUT mult int = 2, OUT a int)
RETURNS RECORD AS $$
    VALUES (mult, a * mult);
$$ language sql;
SELECT * FROM testfoo(mult := 3, a := 14);
```

<table>
<thead>
<tr>
<th>mult</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>42</td>
</tr>
</tbody>
</table>

(1 row)
Function Arguments - argtype

( [[ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )

- argtype (required) (optionally schema-qualified):
  - base, array, composite, or domain types
  - can reference the type of a table column:
    `table_name.column_name%TYPE`
  - Polymorphic "pseudotypes":
    ⇒ anyelement, anyarray, anynonarray, anyenum, anyrange

CREATE FUNCTION testfoo (INOUT a anyelement, INOUT mult anyelement)
RETURNS RECORD AS $$
  VALUES (a * mult, mult);
$$ language sql;
SELECT * FROM testfoo(mult := 3.14, a := 2.71828);  
a | mult
---------
8.5353992 | 3.14
(1 row)
Function Arguments - default_expr

```
( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
```

- default_expr (optional):
  - Used if arg not provided
  - An expression coercible to arg type
  - All input (IN/INOUT/VARIADIC) can have default
  - Following args must also have defaults

CREATE FUNCTION testfoo (IN a int, INOUT mult int = 2, OUT a int)
RETURNS RECORD AS $$
VALUES (mult, a * mult);
$$ language sql;
SELECT * FROM testfoo(14);
```
mult | a
------|
2 | 28
(1 row)
Function Overloading

- Input argument (IN/INOUT/VARIADIC) signature used
- Avoid ambiguities:
  - Type (e.g. REAL vs. DOUBLE PRECISION)
  - Function name same as IN composite field name
  - VARIADIC vs same type scalar

```sql
CREATE OR REPLACE FUNCTION foo (text) RETURNS text AS $$
    SELECT 'Hello ' || $1
$$ LANGUAGE sql;
CREATE OR REPLACE FUNCTION foo (int) RETURNS text AS $$
    SELECT ($1 / 2)::text || ' was here'
$$ LANGUAGE sql;
```

```
SELECT foo('42'), foo(84);
<table>
<thead>
<tr>
<th>foo</th>
<th>foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello 42</td>
<td>42 was here</td>
</tr>
</tbody>
</table>
```

http://www.postgresql.org/docs/9.4/static/xfunc-overload.html

Stephen Frost, Joe Conway
Postgres Open 2014
Function Return Type

[ RETURNS rettype
  | RETURNS TABLE ( column_name column_type [, ...] ) ]

- rettype (required) (optionally schema-qualified):
  - base, array, composite, or domain types
  - can reference the type of a table column:
    table_name.column_name%TYPE
  - Polymorphic "pseudotypes":
    ⇒ anyelement, anyarray, anynonarray, anyenum, anyrange
  - Special "pseudotypes":
    - language_handler: procedural language call handler
    - fdw_handler: foreign-data wrapper handler
    - record: returning an unspecified row type
    - trigger: trigger function
    - void: function returns no value
Function Return Type

```
[ RETURNS rettype
  | RETURNS TABLE ( column_name column_type [, ...] ) ]
```

- **rettype** (required) (optionally schema-qualified):
  - INOUT/OUT args: RETURNS clause may be omitted
    ⇒ Note: does not return a set
  - If RETURNS present, must agree with OUT
  - SETOF modifier - "set returning" or "table" function
CREATE FUNCTION testbar1 (OUT f1 int, OUT f2 text) AS $$
VALUES (42, 'hello'), (64, 'world');
$$ language sql;
SELECT * FROM testbar1();
  f1 | f2
-----+-----
    42 | hello
(1 row)
CREATE FUNCTION testbar2 (OUT f1 int, OUT f2 text)
RETURNS SETOF RECORD AS $$
  VALUES (42, 'hello'), (64, 'world');
$$ language sql;

SELECT * FROM testbar2();
  f1 | f2
-----+-----
  42 | hello
  64 | world
(2 rows)
CREATE TYPE testbar3_type AS (f1 int, f2 text);
CREATE FUNCTION testbar3 ()
RETURNS SETOF testbar3_type AS $$
    VALUES (42, 'hello'), (64, 'world');
$$ language sql;
SELECT * FROM testbar3();
  f1  | f2
------+-----
   42  | hello
   64  | world
(2 rows)
CREATE FUNCTION testbar4 ()
RETURNS TABLE (f1 int, f2 text) AS $$
  VALUES (42, 'hello'), (64, 'world');
$$ language sql;
SELECT * FROM testbar4();

  f1  |  f2
-----+-----
  42  | hello
  64  | world
(2 rows)
CREATE FUNCTION testbar5 ()
RETURNS SETOF RECORD AS $$
    VALUES (42, 'hello'), (64, 'world');
$$ language sql;

SELECT * FROM testbar5() as t(f1 int, f2 text);

<table>
<thead>
<tr>
<th>f1</th>
<th>f2</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>hello</td>
</tr>
<tr>
<td>64</td>
<td>world</td>
</tr>
</tbody>
</table>

(2 rows)
CREATE FUNCTION testbar6 ()
RETURNS SETOF int AS $$
    VALUES (42), (64);
$$ language sql;
SELECT * FROM testbar6();
testbar6
--------
   42
   64
(2 rows)
CREATE FUNCTION testbar7 ()
RETURNS SETOF int AS $$
   VALUES (42), (64);
$$ language sql;
SELECT * FROM testbar7() AS t(f1);
f1
----
  42
  64
(2 rows)
Function Return Type - Targetlist

```sql
SELECT testbar2();
  testbar2
    ----------
     (42,hello)
     (64,world)
(2 rows)
```
Function Return Type - Targetlist, expanded

```sql
SELECT (testbar2()).*;
 f1 | f2
-----+-----
 42 | hello
 64 | world
(2 rows)
```
LANGUAGE

LANGUAGE lang_name

- Language of function body
  - Native: Internal, SQL
  - Interpreted, core: PL/pgSQL, PL/Perl, PL/Python, PL/Tcl
  - Compiled, external: Custom C loadable libraries
  - Some (e.g. perl, tcl) have "trusted" and "untrusted" variants

CREATE FUNCTION ...
LANGUAGE sql;
LANGUAGE plpgsql;
LANGUAGE plperlu;
LANGUAGE plr;
LANGUAGE C;
LANGUAGE internal;
Window Functions

- Indicates function is a window function rather than "normal" function
- Provides ability to calculate across sets of rows related to current row
- Similar to aggregate functions, but does not cause rows to become grouped
- Able to access more than just the current row of the query result
- Window functions can be written in C, PL/R, PL/V8, others?
Window functions built-in

```sql
select distinct proname from pg_proc where proiswindow order by 1;
proname
-------------
cume_dist
dense_rank
first_value
lag
last_value
lead
nth_value
ntile
percent_rank
rank
row_number
(11 rows)
```
Volatility

- **VOLATILE** (default)
  - Each call can return a different result
  - Example: `random()` or `timeofday()`
  - Functions modifying table contents must be declared volatile

- **STABLE**
  - Returns same result for same arguments within single query
  - Example: `now()`
  - Consider configuration settings that affect output

- **IMMUTABLE**
  - Always returns the same result for the same arguments
  - Example: `lower('ABC')`
  - Unaffected by configuration settings
  - Not dependent on table contents
select distinct prono\_name, provolatile
from pg\_proc
where prono\_name in ('lower', 'now', 'timeofday') order by 1;

<table>
<thead>
<tr>
<th>prono_name</th>
<th>provolatile</th>
</tr>
</thead>
<tbody>
<tr>
<td>lower</td>
<td>i</td>
</tr>
<tr>
<td>now</td>
<td>s</td>
</tr>
<tr>
<td>timeofday</td>
<td>v</td>
</tr>
</tbody>
</table>
(3 rows)
**Volatility**

```sql
select lower('ABC'), now(), timeofday() from generate_series(1,3);
```

<table>
<thead>
<tr>
<th>lower</th>
<th>now</th>
<th>timeofday</th>
</tr>
</thead>
<tbody>
<tr>
<td>abc</td>
<td>2014-08-17 12:26:08.407439-07</td>
<td>Sun Aug 17 12:26:08.408005 2014 PDT</td>
</tr>
<tr>
<td>abc</td>
<td>2014-08-17 12:26:08.407439-07</td>
<td>Sun Aug 17 12:26:08.408042 2014 PDT</td>
</tr>
<tr>
<td>abc</td>
<td>2014-08-17 12:26:08.407439-07</td>
<td>Sun Aug 17 12:26:08.408048 2014 PDT</td>
</tr>
</tbody>
</table>

(3 rows)

```sql
select lower('ABC'), now(), timeofday() from generate_series(1,3);
```

<table>
<thead>
<tr>
<th>lower</th>
<th>now</th>
<th>timeofday</th>
</tr>
</thead>
</table>

(3 rows)
Behavior with Null Input Values

- **CALLED ON NULL INPUT** (default)
  - Function called normally with the null input values
- **RETURNS NULL ON NULL INPUT**
  - Function not called when null input values are present
  - Instead, null is returned automatically

```sql
CREATE FUNCTION sum1 (int, int) RETURNS int AS $$
  SELECT $1 + $2
$$ LANGUAGE SQL;
CREATE FUNCTION sum2 (int, int) RETURNS int AS $$
  SELECT COALESCE($1, 0) + COALESCE($2, 0)
$$ LANGUAGE SQL;

SELECT sum1(9, NULL) IS NULL AS "true", sum2(9, NULL);
```

```
true | sum2
-----|------
t  | 9
(1 row)
```
LEAKPROOF requirements

- No side effects
- Reveals no info about args other than by return value
- Planner may push leakproof functions into views created with the security_barrier option
- Can only be set by the superuser
DROP TABLE IF EXISTS all_books CASCADE;
CREATE TABLE all_books(id serial primary key,
    luser text,
    bookname text,
    price int);

INSERT INTO all_books
SELECT g.f,
    CASE WHEN g.f % 2 = 0 THEN 'joe' ELSE 'tom' END,
    'book-' || g.f::text,
    40 + g.f % 20
FROM generate_series(1,8) as g(f);

DROP VIEW IF EXISTS user_books;
CREATE VIEW user_books AS
    SELECT id, luser, bookname, price FROM all_books
    WHERE luser = CURRENT_USER;
GRANT ALL ON user_books TO public;
CREATE OR REPLACE FUNCTION leak_info(text, text) returns int AS $$
BEGIN
    IF $1 != CURRENT_USER THEN
        RAISE NOTICE '%:%', $1, $2;
    END IF;
    RETURN 0;
END;
$$ COST 1 LANGUAGE plpgsql;
EXPLAIN ANALYZE SELECT * FROM user_books WHERE leak_info(luser, bookname) = 0;
NOTICE: tom:book-1
NOTICE: tom:book-3
NOTICE: tom:book-7

QUERY PLAN

Seq Scan on all_books (cost=0.00..1.18 rows=1 width=72) (actual ...
  Filter: ((leak_info(luser, bookname) = 0) AND
           (luser = ("current_user"())::text))
  Rows Removed by Filter: 4
Planning time: 0.674 ms
Execution time: 2.044 ms
(5 rows)
Note the "WITH (security_barrier)" below ...

```
c - postgres
DROP VIEW user_books;
CREATE VIEW user_books WITH (security_barrier) AS
  SELECT id, luser, bookname, price FROM all_books
  WHERE luser = CURRENT_USER;
GRANT ALL ON user_books TO public;
```
Security Attributes - LEAKPROOF

\c - joe
EXPLAIN ANALYZE SELECT * FROM user_books
    WHERE leak_info(luser, bookname) = 0;
QUERY PLAN

Subquery Scan on user_books  (cost=0.00..1.16 rows=1 width=72) (actual ... Filter: (leak_info(user_books.luser, user_books.bookname) = 0)
    ->  Seq Scan on all_books  (cost=0.00..1.14 rows=1 width=72) (actual ... Filter: (luser = ("current_user"()))::text)
          Rows Removed by Filter: 4
Planning time: 0.648 ms
Execution time: 1.903 ms
(7 rows)
Security Attributes - LEAKPROOF

\c - postgres
ALTER FUNCTION leak_info(text, text) LEAKPROOF;

\c - joe
EXPLAIN ANALYZE SELECT * FROM user_books
  WHERE leak_info(luser, bookname) = 0;
NOTICE:  tom:book-1
NOTICE:  tom:book-3
NOTICE:  tom:book-7

QUERY PLAN
---------------------------------------------------------------
  Seq Scan on all_books (cost=0.00..1.18 rows=1 width=72) (actual ...
    Filter: ((leak_info(luser, bookname) = 0) AND
      (luser = ("current_user"())::text))
  Row Removed by Filter: 4
Planning time: 0.646 ms
Execution time: 2.145 ms
(5 rows)
Security Attributes - LEAKPROOF

- **Lesson**
  - Be sure function really is leak proof before making LEAKPROOF

- **Why use LEAKPROOF at all?**
  - Performance (predicate push down)
Security Attributes - SECURITY INVOKER/DEFINER

- **SECURITY INVOKER** (default)
  - Function executed with the rights of the current user
- **SECURITY DEFINER**
  - Executed with rights of creator, like "setuid"

```sql
\c - postgres
CREATE TABLE foo (f1 int);
INSERT INTO foo VALUES(42);
REVOKE ALL ON foo FROM public;
CREATE FUNCTION see_foo() RETURNS TABLE (luser name, f1 int) AS $$
  SELECT CURRENT_USER, * FROM foo
$$ LANGUAGE SQL SECURITY DEFINER;
\c - guest
SELECT * FROM foo;
ERROR: permission denied for relation foo
SELECT CURRENT_USER AS me, luser AS definer, f1 FROM see_foo();
  me  | definer | f1  
+-----+---------+-----+
 guest| postgres| 42  
(1 row)
```

Stephen Frost, Joe Conway  Postgres Open 2014
Optimizer Hints

COST execution_cost
ROWS result_rows

- **execution_cost**
  - Estimated execution cost for the function
  - Positive floating point number
  - Units are cpu_operator_cost
  - Cost is per returned row
  - Default: 1 unit for C-language/internal, 100 units for all others

- **result_rows**
  - Estimated number rows returned
  - Positive floating point number
  - Only allowed when declared to return set
  - Default: 1000

Stephen Frost, Joe Conway
Postgres Open 2014
CREATE FUNCTION testbar8 ()
RETURNS SETOF int AS $$
    VALUES (42), (64);
$$ LANGUAGE sql COST 0.1 ROWS 2;

SELECT procost, prorows FROM pg_proc WHERE proname = 'testbar8';
    procost | prorows
----------+--------
       0.1 |       2
(1 row)
Function Local Configs

```
SET configuration_parameter
{ TO value | = value | FROM CURRENT }
```

- **SET clause**
  - Specified config set to value for duration of function
  - SET FROM CURRENT uses session’s current value

CREATE FUNCTION testbar9 ()
RETURNS SETOF int AS $$
 VALUES (42), (64);
$$ LANGUAGE sql SET work_mem = '512MB';

SELECT proconfig FROM pg_proc WHERE proname = 'testbar9';

---

proconfig

{work_mem=512MB}

(1 row)
Function Body

AS definition
| AS obj_file, link_symbol

definition

- String literal
- Parse by language parser
- Can be internal function name
- Can be path to object file if C language function name matches
- Dollar quote, or escape single quotes and backslashes
Function Body

AS definition
| AS obj_file, link_symbol

- obj_file, link_symbol
  - Used when C language function name does not match SQL function name
  - obj_file is path to object file
    ⇒ $libdir: replaced by package lib dir name, determined at build time
  - link_symbol is name of function in C source code
  - When more than one FUNCTION call refers to same object file, file only loaded once

# pg_config --pkglibdir
/usr/local/pgsql-REL9_4_STABLE/lib
CREATE FUNCTION foobar ()
RETURNS int AS $$
    SELECT 42;
$$ LANGUAGE sql;

CREATE OR REPLACE FUNCTION plr_version ()
RETURNS text
AS '$libdir/plr','plr_version'
LANGUAGE C;
CREATE FUNCTION sum (text, text)
RETURNS text AS $$
    SELECT $1 || ' ' || $2
$$ LANGUAGE SQL;

SELECT sum('hello', 'world');

sum
-------------
hello world
(1 row)
CREATE OPERATOR + (  
    procedure = sum,  
    leftarg = text,  
    rightarg = text
);

SELECT 'hello' + 'world';

----------

hello world
(1 row)
CREATE OR REPLACE FUNCTION concat_ws_comma(text, ANYELEMENT)
RETURNS text AS $$
SELECT concat_ws(',', $1, $2)
$$ LANGUAGE sql;

CREATE AGGREGATE str_agg (ANYELEMENT) (
    sfunc = concat_ws_comma,
    stype = text);

SELECT str_agg(f1) FROM foo;

        str_agg
---------
        41,42
(1 row)
CREATE OR REPLACE FUNCTION sql_with_rows(OUT a int, OUT b text)
RETURNS SETOF RECORD AS $$
    values (1,'a'),(2,'b')
$$ LANGUAGE SQL;

select * from sql_with_rows();

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
</tbody>
</table>

(2 rows)
CREATE TABLE foo (f0 serial, f1 int, f2 text);

CREATE OR REPLACE FUNCTION
sql_insert_returning(INOUT f1 int, INOUT f2 text, OUT id int) AS $$
  INSERT INTO foo(f1, f2) VALUES ($1,$2) RETURNING f1, f2, f0
$$ LANGUAGE SQL;

SELECT * FROM sql_insert_returning(1,'a');
  f1 | f2 | id
  ----+----+----
  1 | a | 1
  (1 row)
CREATE TABLE emp (name text, 
    salary numeric, 
    age integer, 
    cubicle point);

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream 
FROM emp WHERE emp.cubicle ~= point '(2,1)';

SELECT name, 
    double_salary(ROW(name, salary*1.1, age, cubicle)) AS dream 
FROM emp;
CREATE FUNCTION myappend(anyarray, anyelement) RETURNS anyarray AS $$
    SELECT $1 || $2;
$$ LANGUAGE SQL;

SELECT myappend(ARRAY[42,6], 21), myappend(ARRAY['abc','def'], 'xyz');

myappend    |    myappend
-----------+-----------------
   {42,6,21} |   {abc,def,xyz}
(1 row)
CREATE FUNCTION new_emp() RETURNS emp AS $$
    SELECT ROW('None', 1000.0, 25, '(2,2)')::emp;
$$ LANGUAGE SQL;
SELECT new_emp();

--------------------------
new_emp

( None, 1000.0, 25, "(2,2)"

SELECT * FROM new_emp();

name | salary | age | cubicle
-------------------------------
None | 1000.0 | 25 | (2,2)

SELECT (new_emp()).name;

name
------
None
CREATE FUNCTION mleast(VARIADIC numeric[]) RETURNS numeric AS $$
    SELECT min($1[i]) FROM generate_subscripts($1, 1) g(i);
$$ LANGUAGE SQL;

SELECT mleast(10, -1, 5, 4.4);
    mleast
-------
    -1
(1 row)

SELECT mleast(42, 6, 42.42);
    mleast
-------
    6
(1 row)
DEFAULT Arguments

CREATE FUNCTION foo(a int, b int DEFAULT 2, c int DEFAULT 3) RETURNS int LANGUAGE SQL AS $$SELECT $1 + $2 + $3$$;

SELECT foo(10, 20, 30);
    foo
   -----
     60
(1 row)

SELECT foo(10, 20);
    foo
   -----
     33
(1 row)
PL/pgSQL

- PL/pgSQL is SQL plus procedural elements
  - variables
  - if/then/else
  - loops
  - cursors
  - error checking

- Loading the language handler into a database:
  
  ```sql
  CREATE EXTENSION plpgsql;
  ERROR: extension "plpgsql" already exists
  ```

http://www.postgresql.org/docs/9.4/static/plpgsql.html
CREATE OR REPLACE FUNCTION sum (text, text)
RETURNS text AS $$
    BEGIN
        RETURN $1 || ' ' || $2;
    END;
$$ LANGUAGE plpgsql;

SELECT sum('hello', 'world');

sum
-----------
  hello world
(1 row)
 Parameter ALIAS

CREATE OR REPLACE FUNCTION sum (int, int)
RETURNS int AS $$
  DECLARE
    i ALIAS FOR $1;
    j ALIAS FOR $2;
    sum int;
  BEGIN
    sum := i + j;
    RETURN sum;
  END;
$$ LANGUAGE plpgsql;

SELECT sum(41, 1);

    sum
  -----
     42
(1 row)
CREATE OR REPLACE FUNCTION sum (i int, j int) RETURNS int AS $$
DECLARE
    sum int;
BEGIN
    sum := i + j;
    RETURN sum;
END;
$$ LANGUAGE plpgsql;

SELECT sum(41, 1);

    sum
-----
   42
(1 row)
CREATE OR REPLACE FUNCTION even (i int)
RETURNS boolean AS $$
    DECLARE
        tmp int;
    BEGIN
        tmp := i % 2;
        IF tmp = 0 THEN RETURN true;
        ELSE RETURN false;
    END IF;
END;
$$ LANGUAGE plpgsql;

SELECT even(3), even(42);
  even | even
-------|-------
    f   |   t
(1 row)
Control Structures: FOR ... LOOP

CREATE OR REPLACE FUNCTION factorial (i numeric)
RETURNS numeric AS $$
    DECLARE
        tmp numeric; result numeric;
    BEGIN
        result := 1;
        FOR tmp IN 1 .. i LOOP
            result := result * tmp;
        END LOOP;
        RETURN result;
    END;
$$ LANGUAGE plpgsql;
SELECT factorial(42::numeric);

factorial
------------------------------------------------------
1405006117752879898543142606244511569936384000000000000
(1 row)
CREATE OR REPLACE FUNCTION factorial (i numeric)
RETURNS numeric AS $$
    DECLARE tmp numeric; result numeric;
    BEGIN
        result := 1; tmp := 1;
        WHILE tmp <= i LOOP
            result := result * tmp;
            tmp := tmp + 1;
        END LOOP;
        RETURN result;
    END;
$$ LANGUAGE plpgsql;

SELECT factorial(42::numeric);

factorial
-------------------------
14050061177528798985431426062445115699363840000000000
(1 row)
Recursive

CREATE OR REPLACE FUNCTION factorial (i numeric)
RETURNS numeric AS $$
BEGIN
  IF i = 0 THEN
    RETURN 1;
  ELSIF i = 1 THEN
    RETURN 1;
  ELSE
    RETURN i * factorial(i - 1);
  END IF;
END;
$$ LANGUAGE plpgsql;

SELECT factorial(42::numeric);

<table>
<thead>
<tr>
<th>factorial</th>
</tr>
</thead>
<tbody>
<tr>
<td>1405006117752879898543142606244511569936384000000000000</td>
</tr>
</tbody>
</table>

(1 row)
CREATE OR REPLACE FUNCTION format ()
RETURNS text AS $$
DECLARE
tmp RECORD;
BEGIN
    SELECT INTO tmp 1 + 1 AS a, 2 + 2 AS b;
    RETURN 'a = ' || tmp.a || '; b = ' || tmp.b;
END;
$$ LANGUAGE plpgsql;

select format();
   format
-------------
a = 2; b = 4
(1 row)
CREATE OR REPLACE FUNCTION func_w_side_fx() RETURNS void AS
$$
INSERT INTO foo VALUES (41),(42)
$$ LANGUAGE sql;

CREATE OR REPLACE FUNCTION dummy ()
RETURNS text AS $$
BEGIN
    PERFORM func_w_side_fx();
    RETURN 'OK';
END;
$$ LANGUAGE plpgsql;

SELECT dummy();
SELECT * FROM foo;

<table>
<thead>
<tr>
<th>f1</th>
</tr>
</thead>
<tbody>
<tr>
<td>41</td>
</tr>
<tr>
<td>42</td>
</tr>
</tbody>
</table>

(2 rows)
CREATE OR REPLACE FUNCTION get_foo(i int)
RETURNS foo AS $$
DECLARE
  rec RECORD;
BEGIN
  EXECUTE 'SELECT * FROM foo WHERE f1 = ' || i INTO rec;
  RETURN rec;
END;
$$ LANGUAGE plpgsql;

SELECT * FROM get_foo(42);
  f1
----
   42
(1 row)
CREATE OR REPLACE FUNCTION totalbalance()
RETURNS numeric AS $$
    DECLARE
        tmp RECORD; result numeric;
    BEGIN
        result := 0.00;
        FOR tmp IN SELECT * FROM foo LOOP
            result := result + tmp.f1;
        END LOOP;
        RETURN result;
    END;
$$ LANGUAGE plpgsql;

SELECT totalbalance();

     totalbalance
-------------
        83.00
     (1 row)
Error Handling

CREATE OR REPLACE FUNCTION safe_add(a integer, b integer)
RETURNS integer AS $$
BEGIN
    RETURN a + b;
EXCEPTION
    WHEN numeric_value_out_of_range THEN
        -- do some important stuff
        RETURN -1;
    WHEN OTHERS THEN
        -- do some other important stuff
        RETURN -1;
END;
$$ LANGUAGE plpgsql;

http://www.postgresql.org/docs/9.4/static/errcodes-appendix.html
CREATE FUNCTION merge_db(key integer, data text)
RETURNS void AS $$
BEGIN
  LOOP
    UPDATE db SET b = data WHERE a = key;
    IF found THEN RETURN;
    END IF;
  BEGIN
    INSERT INTO db (a, b) VALUES (key, data);
    RETURN;
  EXCEPTION WHEN unique_violation THEN
    -- do nothing
  END;
  END LOOP;
  EXCEPTION WHEN OTHERS THEN
    -- do something else
  END;
$$ LANGUAGE plpgsql;
CREATE TABLE mydata (  
   pk int primary key,  
   mydate date NOT NULL,  
   gender text NOT NULL CHECK(gender IN ('M','F')),  
   mygroup text NOT NULL,  
   id int NOT NULL  
);  

INSERT INTO mydata VALUES  
(1, '2012-03-25','F','A',1),(2, '2005-05-23','F','B',2),  
(3, '2005-09-08','F','B',2),(4, '2005-12-07','F','B',2),  
(5, '2006-02-26','F','C',2),(6, '2006-05-13','F','C',2),  
(7, '2006-09-01','F','C',2),(8, '2006-12-12','F','D',2),  
(9, '2006-02-19','F','D',2),(10, '2006-05-03','F','D',2),  
(11,'2006-04-23','F','D',2),(12,'2007-12-08','F','D',2),  
(13,'2011-03-19','F','D',2),(14,'2007-12-20','M','A',3),  
(15,'2008-06-15','M','A',3),(16,'2008-12-16','M','A',3),  
(17,'2009-06-07','M','B',3),(18,'2009-10-09','M','B',3),  
(19,'2010-01-28','M','B',3),(20,'2007-06-05','M','A',4);
Window Function

```
SELECT id, gender, obs_days, sum(chgd) as num_changes FROM
(SELECT id, gender,
   CASE WHEN row_number() OVER w > 1
       AND mygroup <> lag(mygroup) OVER w THEN 1
       ELSE 0 END AS chgd,
   last_value(mydate) OVER w - first_value(mydate) OVER w AS obs_days
FROM mydata
WINDOW w AS
(PARTITION BY id, gender ORDER BY id, gender, mydate
   ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
) AS ss GROUP BY id, gender, obs_days ORDER BY id, gender;
```

<table>
<thead>
<tr>
<th>id</th>
<th>gender</th>
<th>obs_days</th>
<th>num_changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>2126</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>M</td>
<td>770</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>M</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(4 rows)
### Lateral

```sql
SELECT d.datname, u.rolname, c.config
FROM pg_db_role_setting s
LEFT JOIN pg_authid u ON u.oid = s.setrole
LEFT JOIN pg_database d ON d.oid = s.setdatabase,
LATERAL unnest(s.setconfig) c(config);
```

<table>
<thead>
<tr>
<th>datname</th>
<th>rolname</th>
<th>config</th>
</tr>
</thead>
<tbody>
<tr>
<td>rockstar</td>
<td>work_mem=512MB</td>
<td>search_path=&quot;public, testschema&quot;</td>
</tr>
<tr>
<td>test</td>
<td></td>
<td>work_mem=128MB</td>
</tr>
<tr>
<td>test</td>
<td></td>
<td>statement_timeout=10s</td>
</tr>
<tr>
<td>joe</td>
<td></td>
<td>statement_timeout=60s</td>
</tr>
<tr>
<td>joe</td>
<td></td>
<td>log_min_duration_statement=10s</td>
</tr>
<tr>
<td>joe</td>
<td></td>
<td>maintenance_work_mem=4GB</td>
</tr>
</tbody>
</table>

Stephen Frost, Joe Conway

Postgres Open 2014
Thank You

- Questions?