# PostgreSQL for Developers

Lætitia Avrot

2023/04/12

EDB

# Agenda

- Small things most developers don't know about
- Document-Centric Applications
- Geographic Information Systems (GIS)
- Business Intelligence
- Central Data Center
- Server-Side Languages
- Overview of tools outside Postgres

This talk will cover the advanced features of Postgres that make it the most-loved RDBMS by developers and a great choice for non-relational workloads.

Portions of this presentation were taken from https://momjian.us/presentations

# Small things most developers don't know about

- Postgres is loved by developers but most of them don't even know the full power of Postgres.

# Transactions DDL

```
BEGIN WORK;

ALTER TABLE customer ADD COLUMN debt_limit NUMERIC(10,2);

ALTER TABLE customer ADD COLUMN creation_date TIMESTAMP WITH TIME ZONE;

ALTER TABLE customer RENAME TO cust;

COMMIT;
```

Everything is visible to other transactions only once the `COMMIT` is issued.

# Transactions DDL

```
BEGIN WORK;

EXPLAIN (ANALYZE, BUFFER) [my query];

DROP INDEX …;

EXPLAIN (ANALYZE, BUFFER) [my query];

ROLLBACK;
```

Analyzing how removing an index can impact the plan and duration of the query.

# Arrays

```
CREATE TABLE employee (name TEXT PRIMARY KEY, certifications TEXT[]);

INSERT INTO employee VALUES ('Bill', '{"CCNA", "ACSP", "CISSP"}');

SELECT name
FROM employee
WHERE certifications @> '{ACSP}';

 name
------
 Bill
```

Specific operator to check if an element is in an array.

# Range Types

```
CREATE TABLE car_rental (id SERIAL PRIMARY KEY, time_span TSTZRANGE);

INSERT INTO car_rental
VALUES (DEFAULT, '[2016-05-03 09:00:00, 2016-05-11 12:00:00)');

SELECT * FROM car_rental
WHERE time_span @> '2016-05-09 00:00:00'::timestamptz;

id | time_span
----+-------------------------------------------------------
  1 | ["2016-05-03 09:00:00-04","2016-05-11 12:00:00-04")
```

Only one column to store and query.

# Full Text Search

Specific operator for full text operations

```
SELECT line
FROM fortune
WHERE to_tsvector('english', line) @@ to_tsquery('cat & (sleep | nap)');

                               line
------------------------------------------------------------------
 People who take cat naps don’t usually sleep in a cat’s cradle.
 Q: What is the sound of one cat napping
```

# Trigram Searches

Case insensitive LIKE

```
SELECT line
FROM fortune
WHERE line ILIKE ’%verit%’
ORDER BY 1;

                                  line
--------------------------------------------------------------------------
 body. There hangs from his belt a veritable arsenal of deadly weapons:
 In wine there is truth (In vino veritas).
 Passes wind, water, or out depending upon the severity of the
```

# Regular expressions

Specific operator for regular expressions

```
SELECT line
FROM fortune
WHERE line ~ 'verit'
ORDER BY 1;
```

```
                                  line
-------------------------------------------------------------------------
 body. There hangs from his belt a veritable arsenal of deadly weapons:
 In wine there is truth (In vino veritas).
 Passes wind, water, or out depending upon the severity of the
```

# Regular expressions

| | |
|---|---|
| regexp_count | regexp_replace |
| regexp_instr | regexp_replace |
| regexp_like | regexp_split_to_array |
| regexp_match | regexp_split_to_table |
| regexp_matches | regexp_substr |

# Views and materialized views

- Views = stored SQL query
- Materialized views = stored result of an SQL query

→ Materialized views results might be inconsistent.
→ They have to be refreshed.

# View example

```
CREATE TABLE employee (name TEXT PRIMARY KEY, certifications TEXT[]);
INSERT INTO employee VALUES ('Bill', '{"CCNA", "ACSP", "CISSP"}');

CREATE VIEW acsp AS (SELECT * FROM employee WHERE certifications @> '{ACSP}');


EXPLAIN (SELECT * FROM acsp);

                         QUERY PLAN
---------------------------------------------------------
 Seq Scan on employee  (cost=0.00..1.01 rows=1 width=54)
   Filter: (certifications @> '{ACSP}'::text[])
(2 rows)
```

Scans the employee table with a filter

# Materialized view example

```
CREATE TABLE employee (name TEXT PRIMARY KEY, certifications TEXT[]);
INSERT INTO employee VALUES ('Bill', '{"CCNA", "ACSP", "CISSP"}');

CREATE MATERIALIZED VIEW acsp_m AS
  (SELECT * FROM employee WHERE certifications @> '{ACSP}');


EXPLAIN (SELECT * FROM acsp_m);

                          QUERY PLAN
----------------------------------------------------------
  Seq Scan on ascp_m  (cost=0.00..1.01 rows=1 width=54)
(2 rows)
```

Scans the acsp_m materialized view. No filter.

# Indexes

- Views can't be indexed

```
CREATE INDEX ON acsp(name);
ERROR:  cannot create index on relation "acsp"
DETAIL:  This operation is not supported for views.
```

- Materialized views can be indexed

```
CREATE INDEX ON ascp_m(name);
CREATE INDEX
```

# Refreshing materialized views

- With heavy locks

```
REFRESH MATERIALIZED VIEW ascp_m ;
REFRESH MATERIALIZED VIEW
```

- With light locks

```
REFRESH MATERIALIZED VIEW CONCURRENTLY ascp_m ;
ERROR:  cannot refresh materialized view "public.ascp_m" concurrently
HINT:  Create a unique index with no WHERE clause on one or more columns of
the materialized view.
```

Needs a unique index

# Refreshing materialized views

- With heavy locks

```
REFRESH MATERIALIZED VIEW ascp_m ;
REFRESH MATERIALIZED VIEW
```

- With light locks

```
REFRESH MATERIALIZED VIEW CONCURRENTLY ascp_m ;
ERROR:  cannot refresh materialized view "public.ascp_m" concurrently
HINT:  Create a unique index with no WHERE clause on one or more columns of
the materialized view.
```

Needs a unique index

# Document-Centric Applications

- Postgres was designed from the start to be extensible.  This makes it a great choice for non-relational (No SQL) applications.

# JSON and ANSI SQL - A natural fit

- Fully and naturally integrated  with ANSI SQL in Postgres
- JSON and SQL queries use  the same language, the same planner, and the same ACID compliant transaction framework
- JSON and HSTORE are elegant and easy to use extensions of the underlying object-relational model

# JSON and JSONB

- JSON and JSONB data types are meant to store JSON documents
- JSON will store it as text. It will preserve:
    - White space between tokens
    - The order of keys
    - All keys included duplicates
- JSONB will store a binary representation of the document. It won't preserve
    - White space between tokens
    - The order of keys
    - Duplicate values of a key

- JSON values will be inserted faster
- JSONB values will be queried faster

# JSON Examples

```
CREATE TABLE semi_structured_data (object JSONB);
….
{"name": "Apple Phone", "type": "phone", "brand": "ACME", "price": 200, "available": true,
"warranty_years": 1}
…
INSERT INTO semi_structured_data (object)  VALUES
       (' {        "name": "Apple Phone",
                   "type": "phone",
                   "brand": "ACME",
                   "price": 200,
                   "available": true,
                   "warranty_years": 1
       } ')
```

Create a table JSONB Field

Simple JSON Data Element

Insert this data element into the table `semi_structured_objects`

# JSON Data Type Example

```
{
   "firstName": "John",        // String Type
   "lastName": "Smith",
   "isAlive": true,            // Boolean
   "age": 25,                  // Number Type
   "height_cm": 167.6,
   "address": {                 // Object Type
       "streetAddress": "21 2nd Street",
       "city": "New York",
       "state": "NY",
       "postalCode": "10021-3100"
     },
     "phoneNumbers": [      // Object Array
       {
         "type": "home", "number": "212 555-1234"
       },
       {
         "type": "office", "number": "646 555-4567"
       }
     ],
     "children": [],
     "spouse": null
   }
}
```

# Can store different objects in same field

```
products=# insert into semi_structured_objects values('{ "firstName": "John", "lastName": "Smith", "isAlive": true, "age":
25, "height_cm": 167.6,
   "address": {
     "streetAddress": "21 2nd Street", "city": "New York", "state": "NY", "postalCode": "10021-3100"
   },
   "phoneNumbers": [
     {
       "type": "home",  "number": "212 555-1234"
     },
     {
       "type": "office",  "number": "646 555-4567"
     }
   ],
   "children": [],
   "spouse": null
 }
 ');
```

Different ROWs with different attributes.

```
products=# select * from semi_structured_objects ;
{"name": "Apple Phone", "type": "phone", "brand": "ACME", "price": 200, "available": true, "warranty_years": 1}
{"age": 25, "spouse": null, "address": {"city": "New York", "state": "NY", "postalCode": "10021-3100", "streetAddress": ...
```

# SQL constructs to query the JSON DATA

```
products=# select * from semi_structured_objects ;
{"name": "Apple Phone", "type": "phone", "brand": "ACME", "price": 200, "available": true, "warranty_years": 1}
{"age": 25, "spouse": null, "address": {"city": "New York", "state": "NY", "postalCode": "10021-3100", "streetAddress": ...


products=# select object->>'name' as "Product Name"  from semi_structured_objects where object->>'brand'='ACME';
Product Name
--------------
Apple Phone
(1 row)
products=#
```

Using the operator ->> to select a key

EDB

# Modifying JSON DATA

```
products=# select * from semi_structured_objects ;
{"name": "Apple Phone", "type": "phone", "brand": "ACME", "price": 200, "available": true, "warranty_years": 1}
{"age": 25, "spouse": null, "address": {"city": "New York", "state": "NY", "postalCode": "10021-3100", "streetAddress": ...
```

```
products=# update semi_structured_objects set object['price'] = to_jsonb(150);
```

Using the operator to_jsonb function to format the value as jsonb

```
products=# select * from semi_structured_objects ;
{"name": "Apple Phone", "type": "phone", "brand": "ACME", "price": 150, "available": true, "warranty_years": 1}
{"age": 25, "spouse": null, "address": {"city": "New York", "state": "NY", "postalCode": "10021-3100", "streetAddress": ...
```

# Transform tables to JSON Format

```
products=# select * from product;
weight | sku |     name      | manufacturer | instock |  price
--------+-----+---------------+--------------+---------+---------
   0.5 |   1 | Apple iPhone  | Apple        | t       | $950.00
   0.3 |   2 | Apple Watch   | Apple        | t       | $220.00
   0.2 |   3 | Apple earpods | Apple        | t       | $220.00
     3 |   4 | Macbook Pro   | Apple        | t       | $220.00

products=# select to_json(r) from (select sku as id,  name as  prod_name from product) r;
--------------------------------------
{"id":1,"prod_name":"Apple iPhone"}
{"id":2,"prod_name":"Apple Watch"}
{"id":3,"prod_name":"Apple earpods"}
{"id":4,"prod_name":"Macbook Pro"}
(4 rows)
```

# JSON and ANSI SQL Example

```
 SELECT DISTINCT
      product_type,
      data->>'brand' as Brand,
        data->>'available' as Availability
FROM json_data
JOIN product
ON (product.product_type=semi_structured_objects.object->>'name')
WHERE semi_structured_objects.object->>'available'=true;

 product_type   | brand     | availability
---------------------------+-----------+-------------
 AC3 Phone                | ACME   | true
```

No need for programmatic logic to combine SQL and NoSQL in the application – Postgres does it all

# JSON path

- Support for the SQL/JSON path language in PostgreSQL
- Uses some JavaScript conventions:
  - Dot (.) is used for member access.
  - Square brackets ([]) are used for array access.
  - SQL/JSON arrays are 0-relative, unlike regular SQL arrays that start from 1.
- Variables:
  - $: A variable representing the JSON value being queried
  - $varname : A named variable.
  - @: A variable representing the result of path evaluation in filter expressions

# JSON path - Example

```
{
  "track": {
            "segments": [
            {
            "location":   [ 47.763, 13.4034
],
            "start time": "2018-10-14
10:05:14",
            "HR": 73
            },
            {
            "location":   [ 47.706, 13.2635
],
            "start time": "2018-10-14
10:39:21",
            "HR": 135
            }
            ]
  }
}
```

`$.track.segments:`

retrieves the available track segments

`$.track.segments[*].location:`

retrieves the contents of an array

`$.track.segments[1].location:`

returns the coordinates of the first segment only

`? (condition):`

filters

# JSON path - Example

```
{
  "track": {
            "segments": [
            {
            "location":   [ 47.763, 13.4034
],
            "start time": "2018-10-14
10:05:14",
            "HR": 73
            },
            {
            "location":   [ 47.706, 13.2635
],
            "start time": "2018-10-14
10:39:21",
            "HR": 135
            }
            ]
  }
}
```

```
$.track.segments[*].HR ? (@ > 130)
```

retrieves all heart rate values higher than 130

# JSON path - Example

```
{
  "track": {
            "segments": [
            {
            "location":   [ 47.763, 13.4034
],
            "start time": "2018-10-14
10:05:14",
            "HR": 73
            },
            {
            "location":   [ 47.706, 13.2635
],
            "start time": "2018-10-14
10:39:21",
            "HR": 135
            }
            ]
  }
}
```

```
$.track.segments[*] ?
  (@.location[1] < 13.4).HR ? (@ > 130)
```

First filters all segments by location, and then returns high heart rate values for these segments, if available

# Indexing JSON data

- JSON data are indexed with GIN indexes. They support those operators:
  - `?`, `?|` and `?&`
  - `@>`
  - `@?` and `@@`

```
Example
'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb → t
'{"a":[1,2,3,4,5]}'::jsonb @? '$.a[*] ? (@ > 2)' → t
'{"a":[1,2,3,4,5]}'::jsonb @@ '$.a[*] > 2' → t
'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'd'] → t
'["a", "b", "c"]'::jsonb ?& array['a', 'b'] → t
```

```
CREATE INDEX idxgin ON api USING GIN (jdoc);
```

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"company": "Magnafone"}';
```

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ? 'qui';
```

# More information

- Some consider Postgres JSON support state of the art
- Extensive support for jsonpath
- Webinar by Bruce Momjian, Marc Linster and Thom Brown
    - https://www.youtube.com/watch?v=XsDOMzT1rlo
- Webinar by Andrew Dunstan here:
    - https://www.2ndquadrant.com/en/blog/video-introduction-json-data-types-postgresql/

# Geographic Information Systems (GIS)

- PostGIS is one of the most popular geospatial database offerings in the market. Turning Postgres into one of the most popular and powerful geospatial database is free and simple.

# Power through extensibility — Geo spatial

```
Postgres=# CREATE EXTENSION postgis;
CREATE EXTENSION
```

Now have one of the world’s most popular Geospatial Databases built on well established industry standards.

ArcGIS

Cadcorp
SPATIAL INFORMATION SYSTEM

QGIS

INTERGRAPH

MapInfo

# PostGIS enables a new type of analysis

- What is the largest city with 100 miles of the Grand Canyon?
- How many households are within 1 mile of this fault line?
- If we relocate the office, how does the average commute distance change?
- How many cities within 150KM of Tampa have median income over $50,000.00?
- What truck drove the greatest distance yesterday?

# Can use Spatial AND Traditional analysis tools

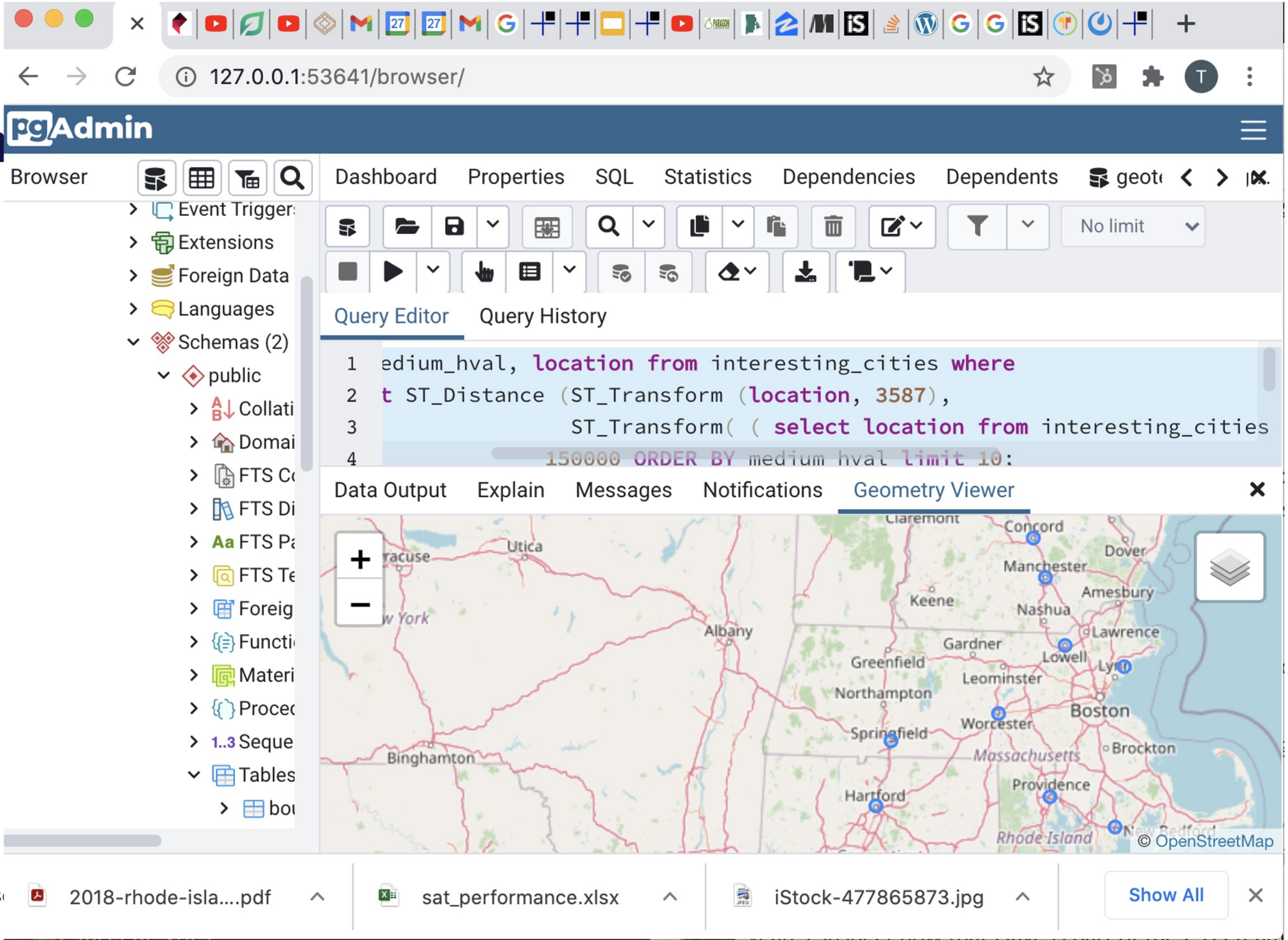
```
-- Interesting cities within 150 KM of Boston with the 10 lowest medium home prices

  select name , medium_hval, location from interesting_cities where
        (select ST_Distance (ST_Transform (location, 3587),
                             ST_Transform( ( select location from interesting_cities
                              where name = 'Boston'), 3587) ) )   < 150000
                              ORDER BY medium_hval limit 10;
```

Spatial Query

Traditional RDBMS expression

EDB

# PgAdm



```
edium_hval, location from interesting_cities where
t ST_Distance (ST_Transform (location, 3587),
                ST_Transform( ( select location from interesting_cities
150000 ORDER BY medium_hval limit 10;
```

# Business Intelligence

- Postgres has advanced functionality for business intelligence.

# Business Intelligence – Advanced CTE Features

Delete a given order,all the items associated with order and place order in a historical table.

```
WITH source (order_id) AS (
    DELETE FROM orders WHERE name = ’my order’ RETURNING order_id
), source2 AS (
    DELETE FROM items USING source WHERE source.order_id =
items.order_id )
INSERT INTO old_orders SELECT order_id FROM source;
```

Variable from first expression passed to next expression. (and again to third expression)

**Less code** to maintain than on any other database
**Fewer round trips** with the server than on any other database

# Business Intelligence — Window Functions

compare each employee's salary with the average salary in his or her department

```
SELECT depname, empno, salary, rank() OVER (PARTITION BY depname ORDER BY salary DESC)
FROM empsalary;

 depname  | empno | salary |          avg
-----------+-------+--------+-----------------------
 develop   |    11 |   5200 | 5020.0000000000000000
 develop   |     7 |   4200 | 5020.0000000000000000
 develop   |     8 |   6000 | 5020.0000000000000000
 develop   |    10 |   5200 | 5020.0000000000000000
 personnel |     5 |   3500 | 3700.0000000000000000
 personnel |     2 |   3900 | 3700.0000000000000000
 sales     |     3 |   4800 | 4866.6666666666666667
 sales     |     1 |   5000 | 4866.6666666666666667
 sales     |     4 |   4800 | 4866.666666666666667
(9 rows)
```

# Business Intelligence — Advanced value expressions

Compare total with count of subsets.

```
SELECT count(*) count_all,
         count(*) FILTER(WHERE bid=1) count_1,
         count(*) FILTER(WHERE bid=2) count_2
  FROM pgbench_history;
```

```
 count_all | count_1 | count_2
-----------+---------+---------
      7914 |     758 |     784

(1 row)
```

# Business Intelligence — Specialized Indexes

Specialized Indexes for all data types and access patterns

| **Index Type** | **Optimized For** |
|---|---|
| B-Tree | Range queries with low selectivity and largely unique values. The traditional database index. |
| Special ops (text_pattern_ops) for B-Tree | LIKE operations |
| BRIN | Time series data, multi-terabyte tables |
| HASH | Equality lookups on large datasets (key / value store) use cases. |
| GiST | Unstructured Data i.e. Geo Spatial Types |
| GIN | JSON Data, Full Text Search, JSONB Data |
| SP-GiST | SP-GıST is ideal for indexes whose keys have many duplicate prefixes |

EDB

# Business Intelligence — Specialized Indexes

Specialized Indexes for non-relational data

| Index Type | Optimized For |
|---|---|
| PARTIAL | When only a specific set of values will be looked up |
| COVERING | For access patterns to unindex values navigated to by an index. |
| EXPRESSION | Allow for variances in keys |

# Partitioning

- PostgreSQL supports
  - Range Partitioning
  - List Partitioning
  - Hash Partitioning
- Needs a partition key
- Allows subpartitioning
- Performance will only improves if:
  - We don't retrieve all data
  - The partition key is part of the WHERE clause

# Partitioning

- PostgreSQL supports
  - Range Partitioning
  - List Partitioning
  - Hash Partitioning
- Partitioning needs a partition key

- PostgreSQL support subpartitioning
- Performance will only improves if
  - We need to retrieve data from a few partitions only
  - The partition key is part of the WHERE clause

# Partitioning - limitations

1. Unique constraints on partitioned tables must include all the partition key columns. One work-around is to create unique constraints on each partition instead of a partitioned table.
2. Partition does not support BEFORE ROW triggers on partitioned tables. If necessary, they must be defined on individual partitions, not the partitioned table.
3. Range partition does not allow NULL values.

# Central Data Center

- Postgres can function as a central integration point for your data center using Foreign Data Wrappers.

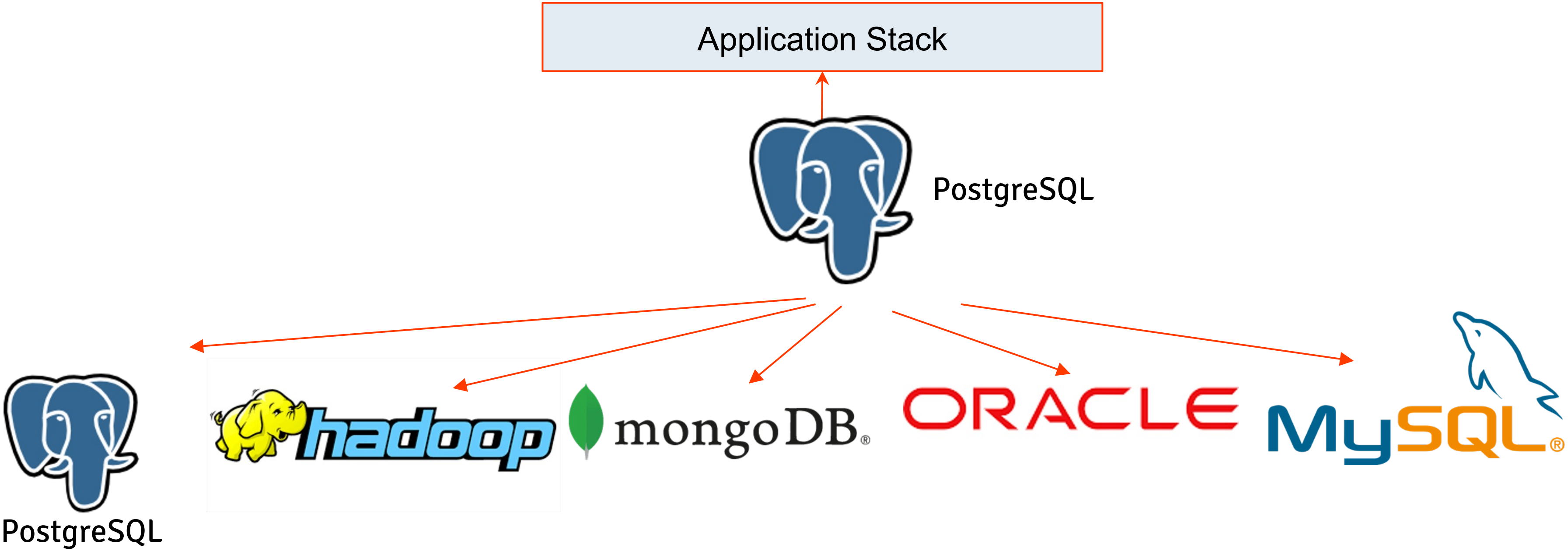
# Power through extensibility — Foreign Data Wrappers

```
postgres=# CREATE EXTENSION postgres_fdw;
CREATE EXTENSION
```

# Foreign Data Wrappers

```
CREATE SERVER postgres_server FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host ’localhost’, dbname
’fdw_test’);

create SERVER oracle_server foreign data wrapper oracle_fdw options (dbserver
'//<oracle_servefr_IP>/<sid>' );

CREATE SERVER mongo_server FOREIGN DATA WRAPPER mongo_fdw OPTIONS (address '127.0.0.1', port '27017');

CREATE SERVER hadoop_server FOREIGN DATA WRAPPER hdfs_fdw  OPTIONS (host '127.0.0.1');

CREATE SERVER mysql_server FOREIGN DATA WRAPPER mysql_fdw OPTIONS (host '127.0.0.1', port '3306');
```

# Foreign Data Wrapper Access



# Server-Side Languages

- Postgres has server-side language support for almost all developers.

# Server-Side Programming Languages

- PL/Java
- PL/Python
- PL/R
- PL/pgSQL (like PL/SQL)
- PL/Ruby
- PL/Scheme
- PL/sh
- PL/Tcl
- PL/v8 (JavaScript)
- SPI (C)

```
CREATE LANGUAGE plpython3u;

CREATE OR REPLACE FUNCTION pymax (a integer,
b integer) RETURNS integer AS
   $$
       if a > b:
         return a
       return b
   $$ LANGUAGE plpython3u;

SELECT pymax(12, 3);

pymax
 -------
     12
 (1 row)
```

# Thank You!

EDB