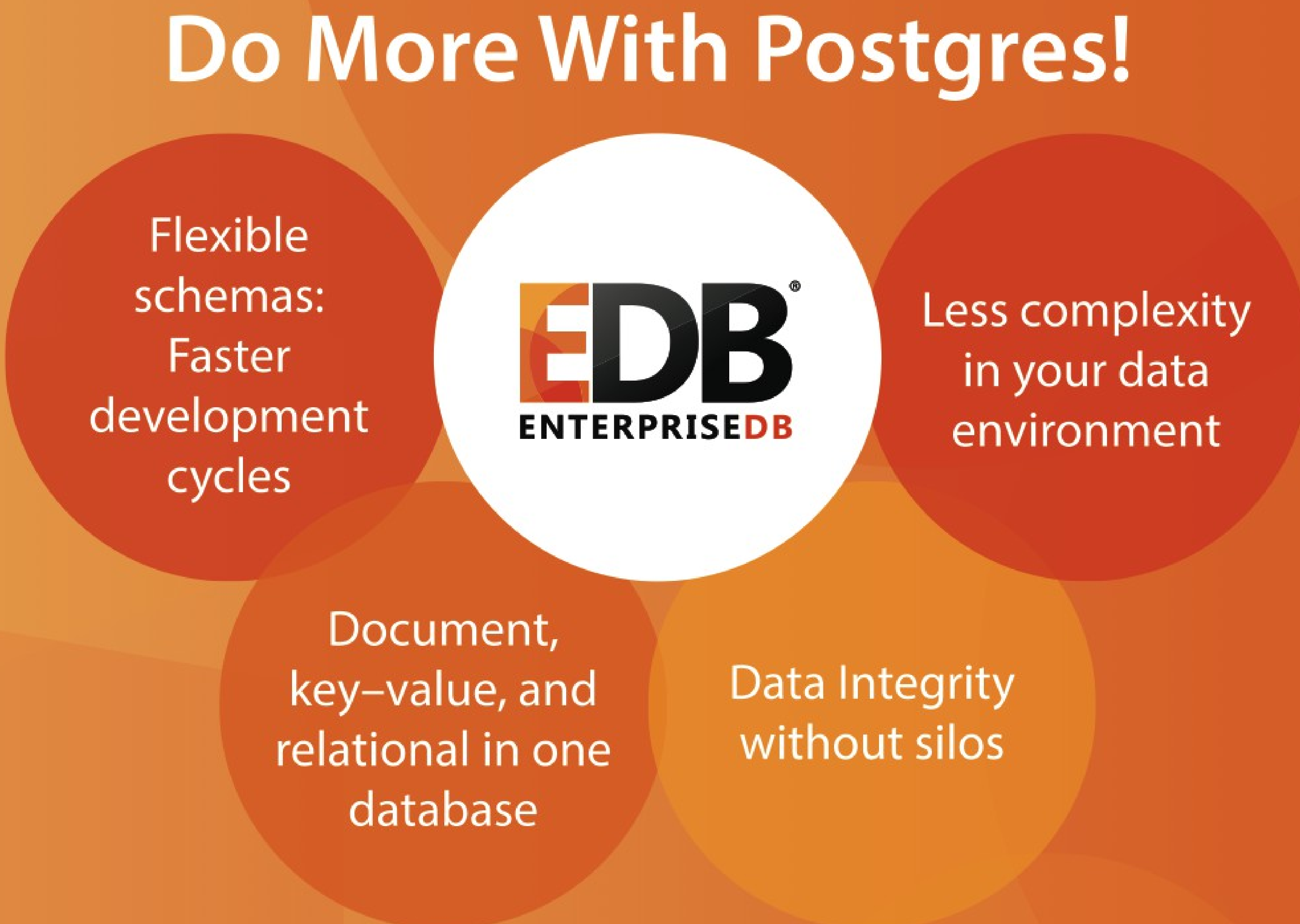


NoSQL On ACID

October 21, 2014

Let's Ask Ourselves, Why NoSQL?

- `Where did NoSQL come from?`
  - Where all cool tech stuff comes from – Internet companies
- `Why did they make NoSQL?`
  - To support huge data volumes and evolving demands for ways to work with new data types
- `What does NoSQL accomplish?`
  - Enables you to work with new data types: email, mobile interactions, machine data, social connections
  - Enables you to work in new ways: incremental development and continuous release
- `Why did they have to build something new?`
  - There were limitations to most relational databases

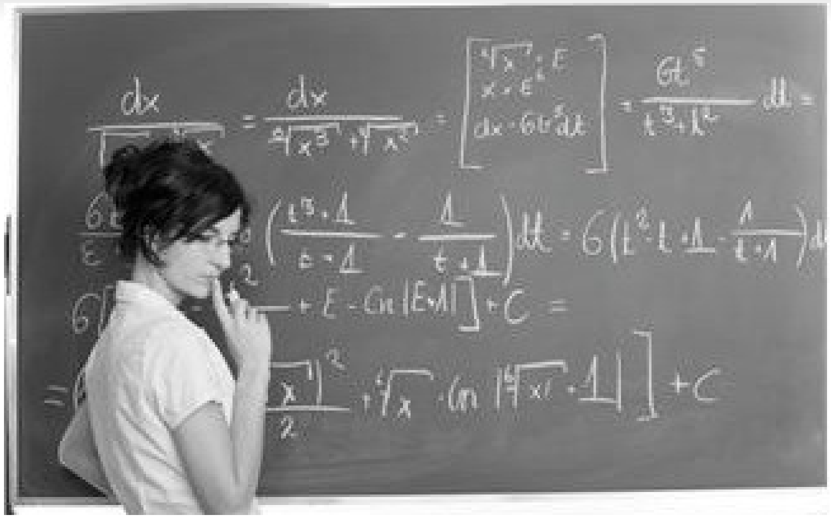
EDB ENTERPRISEDB

NoSQL: Real-world Applications

- Emergency Management System
  - High variability among data sources required high schema flexibility
- Massively Open Online Course
  - Massive read scalability, content integration, low latency
- Patient Data and Prescription Records
  - Efficient write scalability
- Social Marketing Analytics
  - Map reduce analytical approaches

Source: Gartner, A Tour of NoSQL in 8 Use Cases,
by Nick Heudecker and Merv Adrian, February 28, 2014

Postgres' Response



- HSTORE
  - Key-value pair
  - Simple, fast and easy
  - Postgres v 8.2 – pre-dates many NoSQL-only solutions
  - Ideal for flat data structures that are sparsely populated
- JSON
  - Hierarchical document model
  - Introduced in Postgres 9.2, perfected in 9.3
- JSONB
  - Binary version of JSON
  - Faster, more operators and even more robust
  - Postgres 9.4

Postgres: Key-value Store

- Supported since 2006, the HStore contrib module enables storing key/value pairs within a single column
- Allows you to create a schema-less, ACID compliant data store within Postgres

- Create single HStore column and include, for each row, only those keys which pertain to the record
- Add attributes to a table and query without advance planning

•Combines flexibility with ACID compliance

HSTORE Examples

- Create a table with HSTORE field

```
CREATE TABLE hstore_data (data HSTORE);
```

- Insert a record into hstore_data

```
INSERT INTO hstore_data (data) VALUES (’
          "cost"=>"500",
          "product"=>"iphone",
          "provider"=>"apple"');
```

- Select data from hstore_data

```
SELECT data FROM hstore_data ;
------------------------------------------
"cost"=>"500”,"product"=>"iphone”,"provider"=>"Apple"
(1 row)
```

Postgres: Document Store

- JSON is the most popular data-interchange format on the web
- Derived from the ECMAScript Programming Language Standard (European Computer Manufacturers Association).
- Supported by virtually every programming language
- New supporting technologies continue to expand JSON's utility
  - PL/V8 JavaScript extension
  - Node.js
- Postgres has a native JSON data type (v9.2) and a JSON parser and a variety of JSON functions (v9.3)
- Postgres will have a JSONB data type with binary storage and indexing (coming – v9.4)

Why JSON

- Wherever is JAVA Script. especially Browser.
- Most of Languages Support it.
- Node.Js is becoming popular.
- Lighter and more compact than XML.
- Most application don't need richer structure like XML.
- Flexible Structure.
- Due to its flexible Structure, good data type for NoSQL.

JSON Examples

- Creating a table with a JSONB field

```
CREATE TABLE json_data (data JSONB);
```

- Simple JSON data element:

```
{"name": "Apple Phone", "type": "phone", "brand":
"ACME", "price": 200, "available": true,
"warranty_years": 1}
```

- Inserting this data element into the table json_data

```
INSERT INTO json_data (data) VALUES
(' {"name": "Apple Phone",
"type": "phone",
"brand": "ACME",
"price": 200,
"available": true,
"warranty_years": 1
} ');
```

JSON Examples

- JSON data element with nesting:

```
{"full name": "John Joseph Carl Salinger",
"names":
   [
    {"type": "firstname", "value": "John"},
    {"type": "middlename", "value": "Joseph"},
    {"type": "middlename", "value": "Carl"},
    {"type": "lastname", "value": "Salinger"}
   ]
}
```

A simple query for JSON data

```
SELECT DISTINCT
data->>'name' as products
FROM json_data;

            products
------------------------------
 Cable TV Basic Service Package
 AC3 Case Black
 Phone Service Basic Plan
 AC3 Phone
 AC3 Case Green
 Phone Service Family Plan
 AC3 Case Red
 AC7 Phone
```

This query does not return JSON data – it returns text values associated with the key ‘name’

A query that returns JSON data

```
SELECT data FROM json_data;

data

------------------------------------------

 {"name": "Apple Phone", "type": "phone", "brand":
"ACME", "price": 200, "available": true,
"warranty_years": 1}
```

This query returns the JSON data in its original format

JSON is defined per RFC – 7159
For more detail please refer
http://tools.ietf.org/html/rfc7159

JSON Data Types

- `1. Number:`
  - Signed decimal number that may contain a fractional part and may use exponential notation.
  - No distinction between integer and floating-point
- `2. String`
  - A sequence of zero or more Unicode characters.
  - Strings are delimited with double-quotation mark
  - Supports a backslash escaping syntax.
- `3. Boolean`
  - Either of the values true or false.
- `4. Array`
  - An ordered list of zero or more values,
  - Each values may be of any type.
  - Arrays use square bracket notation with elements being comma-separated.
- `5. Object`
  - An unordered associative array (name/value pairs).
  - Objects are delimited with curly brackets
  - Commas to separate each pair
  - Each pair the colon ':' character separates the key or name from its value.
  - All keys must be strings and should be distinct from each other within that object.
- `6. null`
  - An empty value, using the word null

JSON Data Type Example

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

History of JSON in PostgreSQL

History: JSON – Before 9.2

- JSON could only be stored as simple text.
- Did not have structure Validation.
- Did not have Supported functions/operated
- Application had to do most of work for
  - Validation
  - Verification
  - Extraction

History: JSON – In 9.2

- New data type JSON.
- Data can also be stored as text.
- Validate stored value is valid JSON.
- Proved following two supported functions:
  - array_to_json(anyarray [, pretty_bool])
  - row_to_json(record [, pretty_bool])
- Missing feature:
  - JSON processing was missing
  - User has to use PLV8, PLPerl etc..

History: JSON – In 9.3

- Add operators and functions to extract elements from JSON values
  - Allow JSON values to be converted into records.
  - Add functions to convert scalars, records, and hstore values to JSON
- Functions honour casts to JSON for non built-in types.
- New functions for HSTORE to JSON
  - hstore_to_json(hstore)
  - hstore_to_json_loose(hstore).
- Parser exposed for use by other modules such as extensions as an API.

Operators and Functions

- `extraction operators:`
  - `->` fetch an array element or object member as json
  - json arrays are 0 based, unlike SQL arrays
  - `'[4,5,6]'::json->2` $\Longrightarrow$ `6`
  - `'{"a":1,"b":2}'::json->'b'` $\Longrightarrow$ `2`
- `9.3 extraction operators:`
  - ->> fetch an array element or object member as text
  - `'["a","b","c"]'::json->2` $\Longrightarrow$ `c`
  - Instead of "c"

Operators and Functions

- JSON Extraction Functions:
  - `json_extract_path(json, VARIADIC path_elems text[]);`
  - `json_extract_path_text(json, VARIADIC path_elems text[]);`

- Same as #> and #>> operators, but you can pass the path as a variadic array
- `json_extract_path('{"a":[6,7,8]}','a','1')` ⟹7

Operators and Functions

- 9.3 turn JSON into records:

- CREATE TYPE x AS (a int, b int);
- SELECT * FROM json_populate_record(null::x, '{"a":1,"b":2}', false);
- SELECT * FROM json_populate_recordset(null::x,'[{"a":1," b":2},{"a":3,"b":4}]', false);

Operators and Functions

- 9.3 turn JSON into key/value pairs
  - `SELECT * FROM json_each('{"a":1,"b":"foo"}')`
  - `SELECT * FROM json_each_text('{"a":1,"b":"foo"}')`

- Deliver columns named “key” and “value”

Operators and Functions

- 9.3 get keys from JSON object:
- SELECT * FROM json_object_keys('{"a":1,"b":"foo"}')

- 9.3 JSON array processing:
- SELECT json_array_length('[1,2,3,4]');
- SELECT * FROM json_array_elements('[1,2,3,4]')

JSON 9.4 – New Operators and Functions

- JSON
  - New JSON creation functions (json_build_object, json_build_array)
  - json_typeof – returns text data type (‘number’, ‘boolean’, …)
- JSONB data type
  - Canonical representation
    - Whitespace and punctuation dissolved away
    - Only one value per object key is kept
    - Last insert wins
    - Key order determined by length, then bytewise comparison
  - Equality, containment and key/element presence tests
  - New JSONB creation functions
  - Smaller, faster GIN indexes
  - jsonb subdocument indexes
    - Use “get” operators to construct expression indexes on subdocument:
    - `CREATE INDEX author_index ON books USING GIN ((jsondata -> 'authors'));`
    - `SELECT * FROM books WHERE jsondata -> 'authors' ? 'Carl Bernstein'`

9.4 Features Set:

- New json creation functions
- New utility functions
- jsonb type
- Efficient operations Indexable Canonical

9.4 Features – new json aggregate

- json_object_agg(“any”, “any”)
- Turn a set of key value pairs into a json object
- SELECT json_object_agg(name, population) from cities;
  - { “Smallville”: 300, “Metropolis”: 1000000}

9.4 Features – json creation functions

- json_build_object(VARIADIC “any”)
- json_build_array(VARIADIC “any”)
- json_object(text[])
- json_object(keys text[], values text[])

9.4 Features – json creation functions (Examples)

- `SELECT json_build_object('a',1,'b',true)`
  - `{“a”: 1, “b”: true}`
- `SELECT json_build_array('a',1,'b',true)`
  - `[“a”, 1, “b”, true]`
- `SELECT json_object(array['a','b','c','d']`
- `Or SELECT json_object(array[['a','b'],['c','d']]`
- `Or SELECT json_object(array['a','c'],array['b','d'])`
  - `{“a”:”b”, “c”:”d”}`

9.4 features – json_typeof

- json_typeof(json) returns text Result is one of:
  - 'object’
  - 'array’
  - 'string’
  - 'number'
  - 'boolean'
  - 'null’
  - Null

9.4 features – jsonb type

- Accepts the same inputs as json
- Uses the 9.3 parsing API
- Checks Unicode escapes, especially use of surrogate pairs, more thoroughly than json.
- Representation closely mirrors json syntax

9.4 Features – jsonb canonical representation

- Whitespace and punctuation dissolved away
- Only one value per object key is kept
- Last one wins.
- Key order determined by length, then bytewise comparison

9.4 Features – jsonb operators

- Has the json operators with the same semantics:
  - -> ->> #> #>>
- Has standard equality and inequality operators
  - = <> > < >= <=
- Has new operations testing containment, key/element presence
  - @> <@ ? ?| ?&

9.4 Features – jsonb equality and inequality

- Comparison is piecewise
  - Object > Array > Boolean > Number > String > Null
    Object with n pairs > object with n - 1 pairs
- Array with n elements > array with n - 1 elements
- Not particularly intuitive
- Not ECMA standard ordering, which is possibly not suitable anyway

9.4 features – jsonb functions

- jsonb has all the json processing functions, with the same semantics
- i.e. functions that take json arguments
- Function names start with jsonb_ instead of json_

- jsonb does not have any of the json creation functions
- i.e. functions that take non-json arguments and output json
- Workaround: cast result to jsonb

9.4 features – jsonb indexing

- `2 ops classes for GIN indexes`
- `Default supports contains and exists operators:`
  - @> ? ?& ?|
- `Non-default ops class jsonb_path_ops only supports`
  - @> operator
  - Faster
  - Smaller indexes

9.4 features – jsonb subdocument indexes

- Use “get” operators to construct expression indexes on subdocument:

- `CREATE INDEX author_index ON books USING GIN ((jsondata -> 'authors'));`
- `SELECT * FROM books WHERE jsondata -> 'authors' ? 'Carl Bernstein'`

PLV8

Java Script Language In database

PLV8: V8 Engine Java Script language

- PLV8 is a shared library that provides a PostgreSQL procedural language powered by
- V8 JavaScript Engine.
- Language you can write in your JavaScript function that is callable from SQL.

PLV8: Installation

- Requires g++ version 4.5.1 or 4.4.x
- For Installation of PLV8, we need V8 engine on server
  - V8 JavaScript Engine is an open source JavaScript engine developed by Google for the Google Chrome web browser.
- To install V8, you can use RPMS:
  - v8-devel-3.14.5.10-9.el6.x86_64
  - v8-3.14.5.10-9.el6.x86_64
- OR
- Using source code.

PLV8: Installation

- `cd ~/build`
- `git clone https://code.google.com/p/plv8js/`
- `cd plv8js`
- `make`
- `make install`
- `psql -d` *`dbname`* `-c "CREATE EXTENSION plv8"`

PLV8: Examples

```
CREATE OR REPLACE FUNCTION plv8_test(keys text[], vals
text[]) RETURNS
  text AS $$
  var o = {};
  for(var i=0; i<keys.length; i++){
    o[keys[i]] = vals[i];
  }
  return JSON.stringify(o);
$$ LANGUAGE plv8 IMMUTABLE STRICT;
```

- `SELECT plv8_test(ARRAY['name', 'age'], ARRAY['Tom', '29']);`

PLV8: Examples

```
CREATE TYPE rec AS (i integer, t text);
CREATE FUNCTION set_of_records() RETURNS SETOF rec AS
$$
// plv8.return_next() stores records in an internal tuplestore,
// and return all of them at the end of function.
    plv8.return_next( { "i": 1, "t": "a" } );
    plv8.return_next( { "i": 2, "t": "b" } );

 // You can also return records with an array of JSON.
    return [ { "i": 3, "t": "c" }, { "i": 4, "t": "d" } ];
$$
LANGUAGE plv8;
```

PLV8: Examples

```
SELECT * FROM set_of_records();
 i | t
---+---
 1 | a
 2 | b
 3 | c
 4 | d
(4 rows)
```

PLV8: Built in functions

- `plv8.elog( elevel, ... )`
- Function print messages to server and/or client logs just like as RAISE in PL/pgSQL
- Acceptable elevels are
  - DEBUG[1-5],
  - LOG,
  - INFO,
  - NOTICE,
  - WARNING and
  - ERROR.

PLV8: Built in functions

- `plv8.execute( sql [, args] )`
- Execute SQL statements and retrieve the result. "args" is an optional argument that replaces $n placeholders in "sql".
- Example:
- `var json_result = plv8.execute( 'SELECT * FROM tbl' );`
- `var num_affected = plv8.execute( 'DELETE FROM tbl WHERE price > $1', [ 1000 ] );`

PLV8: Built in functions

- `plv8.prepare( sql, [, typenames] )`
- Create a prepared statement. The *typename* parameter is an array where each element is a string to indicate PostgreSQL type name for bind parameters. Returned value is an object of PreparedPlan.
- object must be freed by plan.free() before leaving the function.
- Example:
- `var plan = plv8.prepare( 'SELECT * FROM tbl WHERE col = $1', ['int'] );`
- `var rows = plan.execute( [1] );`

PLV8: Built in functions

- `PreparedPlan.execute( [args] )`
- *args* parameter is as plv8.execute(), and
- can be omitted if the statement doesn't have parameters at all.
- The result of this method is same as in plv8.execute().

PLV8: Built in functions

- `PreparedPlan.cursor( [args] )`
- Open a cursor from the prepared statement.
- *args* parameter is as plv8.execute(), and
- can be omitted if the statement doesn't have parameters at all.
- The returned object is of Cursor.
- It must be closed by Cursor.close() before leaving the function.

PLV8: Built in functions

```
PreparedPlan.cursor( [args] )


var plan = plv8.prepare( 'SELECT * FROM tbl WHERE col = $1',
['int'] );
var cursor = plan.cursor( [1] );
var sum = 0, row;
while (row = cursor.fetch()) {
  sum += row.num;
}
cursor.close();
plan.free();
return sum;
```

PLV8: Built in functions

- `PreparedPlan.free()`
  - Free the prepared statement.
- `Cursor.fetch()`
  - Fetch a row from the cursor and return as an object (note: not an array.) Fetching more than one row, and move() aren't currently implemented.
- `Cursor.close()`
  - Close the cursor.

PLV8: Built in functions

- `plv8.subtransaction( func )`
- Function runs the argument function within a sub-transaction.
- Needed when you want multiple “execute(query)” commands to be run atomically.
- If one of the statements fails then everything which is run in this function will be rolled back.
- **Note:** if an exception is thrown from the subtransaction function, the exception goes out of subtransaction(), so you'll typically need another try-catch block outside.

PLV8: Built in functions

- `plv8.subtransaction( func )`
- Example:

```
try{
  plv8.subtransaction(function(){
    plv8.execute("INSERT INTO tbl VALUES(1)");  -- should be rolled back!
    plv8.execute("INSERT INTO tbl VALUES(1/0)");-- occurs an exception
  });
} catch(e) {
  ... do fall back plan ...
}
```

JSON and ANSI SQL - PB&J for the DBA

- JSON is 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 ANSI SQL Example

ANSI SQL

JSON

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

Bridging between SQL and JSON

Simple ANSI SQL Table Definition

```
CREATE TABLE products (id integer, product_name text );
```

Select query returning standard data set

```
SELECT * FROM products;

 id | product_name
----+--------------
  1 | iPhone
  2 | Samsung
  3 | Nokia
```

Select query returning the same result as a JSON data set

```
SELECT ROW_TO_JSON(products) FROM products;

{"id":1,"product_name":"iPhone"}
{"id":2,"product_name":"Samsung"}
{"id":3,"product_name":"Nokia”}
```

JSON and BSON

- `BSON – stands for 'Binary JSON'`
- `BSON != JSONB`
  - BSON cannot represent an integer or floating-point number with more than 64 bits of precision.
  - JSONB can represent arbitrary JSON values.
- `Caveat Emptor!`
  - This limitation will not be obvious during early stages of a project!

JSON, JSONB or HSTORE?

- JSON/JSONB is more versatile than HSTORE
- HSTORE provides more structure
- JSON or JSONB?
  - if you need any of the following, use JSON
    - Storage of validated json, without processing or indexing it
    - Preservation of white space in json text
    - Preservation of object key order Preservation of duplicate object keys
    - Maximum input/output speed
- For any other case, use JSONB

JSONB and Node.js - Easy as π

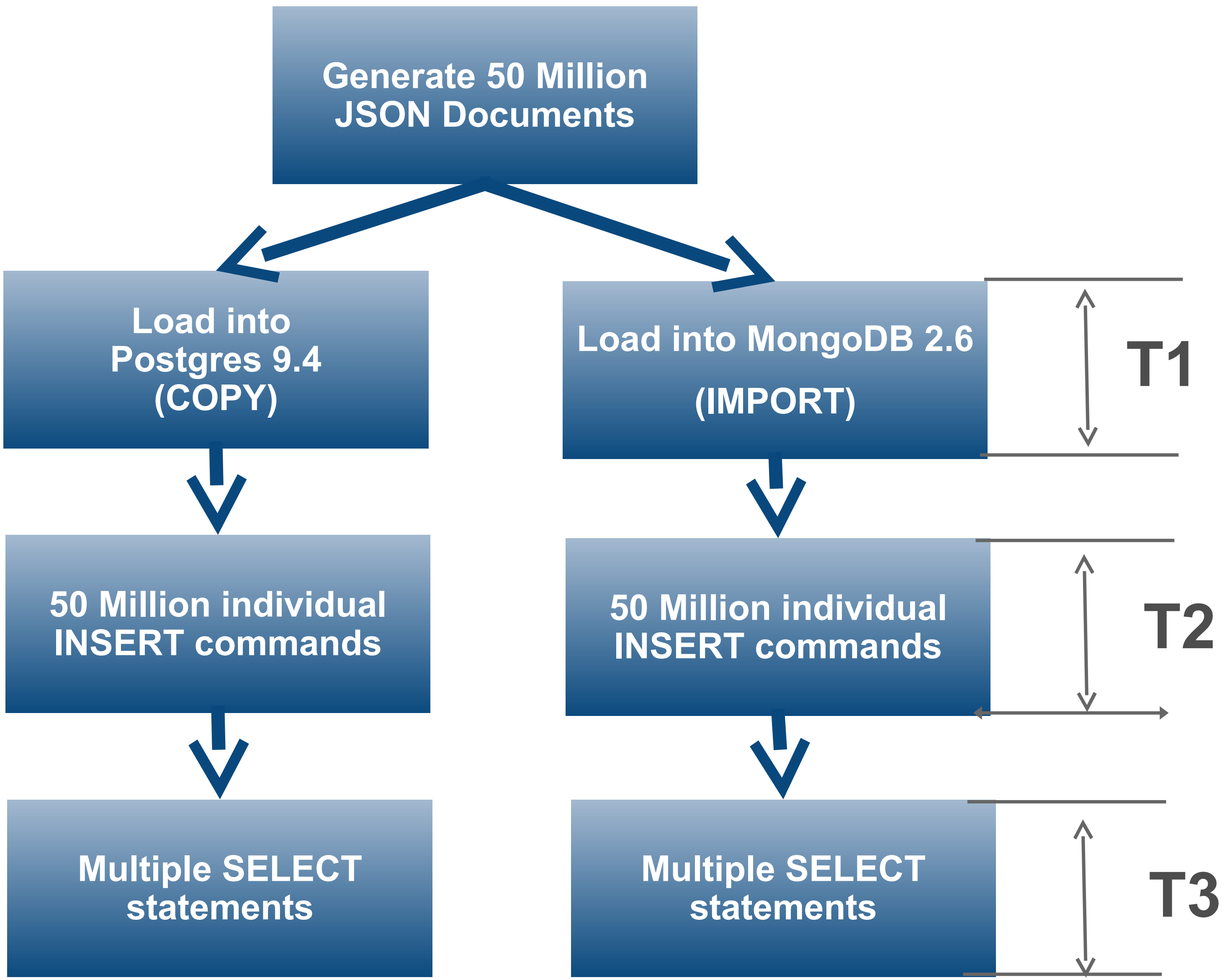
```
// require the Postgres connector
var pg = require("pg");

// connection to local database
var conString = "pg://postgres:password@localhost:5432/nodetraining";

var client = new pg.Client(conString);
client.connect();

// initiate the sample database
 client.query("CREATE TABLE IF NOT EXISTS emps(data jsonb)");
 client.query("TRUNCATE TABLE emps;");
 client.query('INSERT INTO emps VALUES($JSON$ {"firstname": "Ronald" , "lastname":"McDonald" }$JSON$)')
 client.query('INSERT  INTO emps values($JSON$ {"firstname": "Mayor", "lastname": "McCheese"}$JSON$)')

// run SELECT query
 client.query("SELECT * FROM emps",function(err,result){
     console.log("Test Output of JSON Result Object");
     console.log(result);
     console.log("Parsed rows");

// parse the result set
    for (var i = 0; i< result.rows.length ; i++ ){
        var data = JSON.parse(result.rows[i].data);
        console.log("First Name => "+ data.firstname + "\t| Last Name => " + data.lastname);
    }
 client.end();
})
```

JSON Performance Evaluation

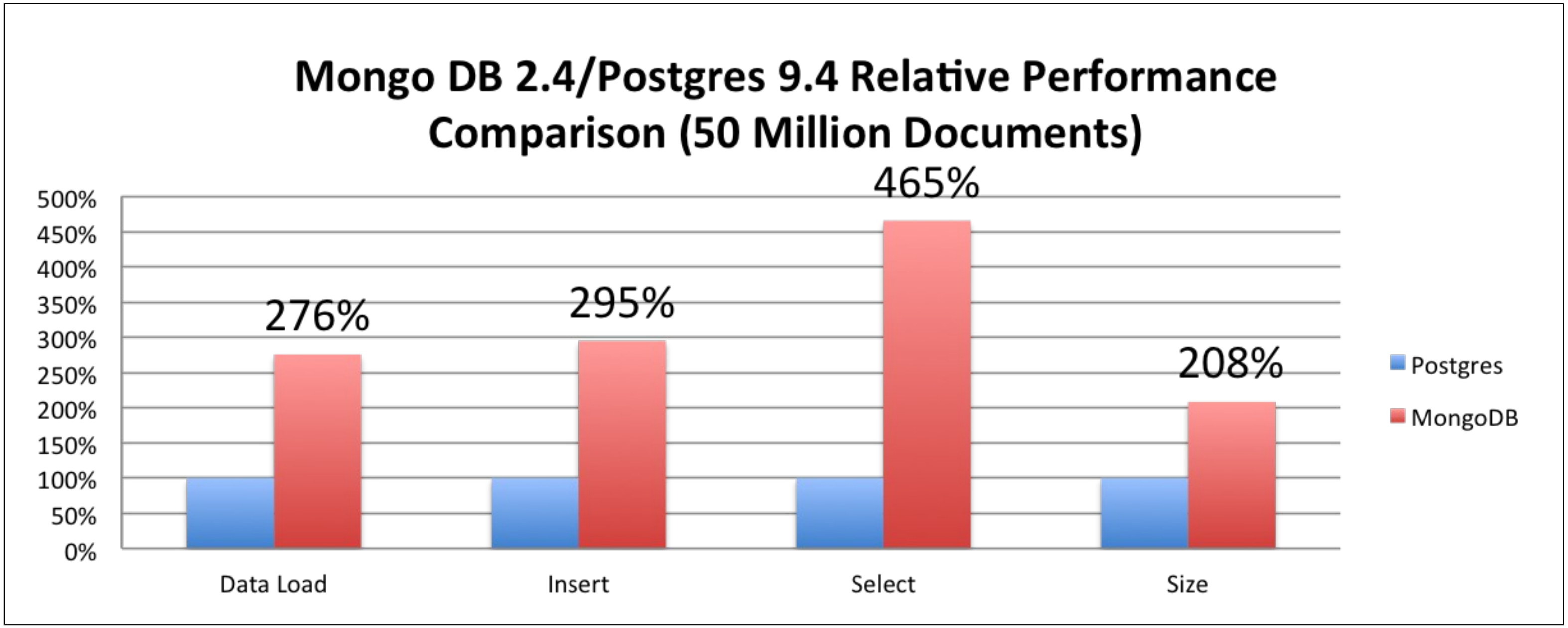
- Goal
  - Help our customers understand when to chose Postgres and when to chose a specialty solution
  - Help us understand where the NoSQL limits of Postgres are
- Setup
  - Compare Postgres 9.4 to Mongo 2.6
  - Single instance setup on AWS M3.2XLARGE (32GB)
- Test Focus
  - Data ingestion (bulk and individual)
  - Data retrieval

EDB ENTERPRISEDB

Performance Evaluation



NoSQL Performance Evaluation



	Postgres	MongoDB
Data Load (s)	4,732	13,046
Insert (s)	29,236	86,253
Select (s)	594	2,763
Size (GB)	69	145

Correction to earlier versions:

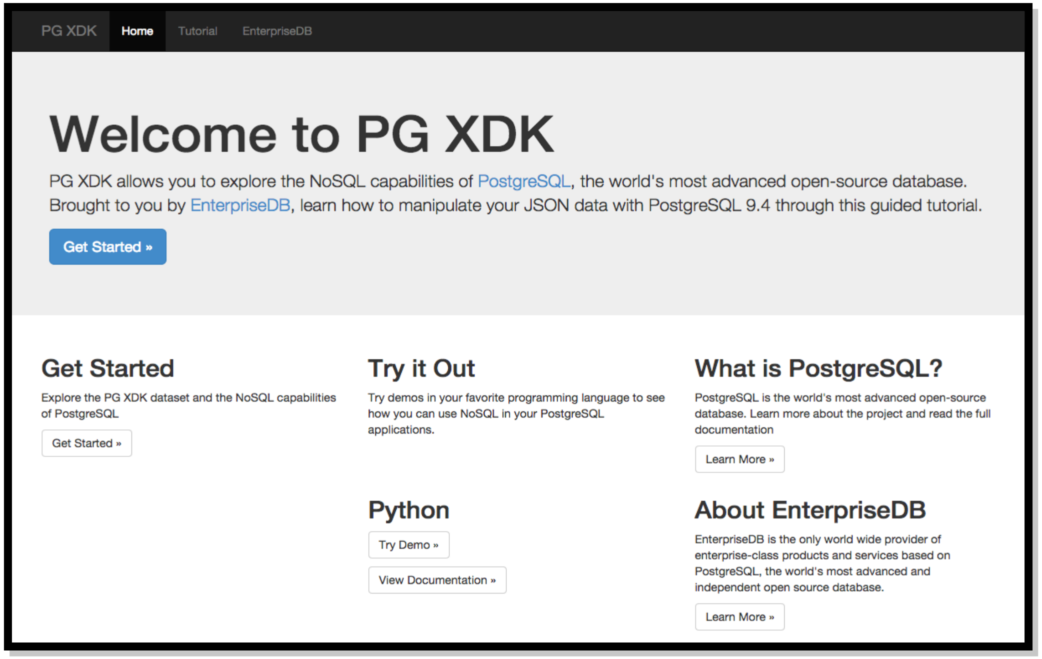
MongoDB console does not allow for INSERT of documents > 4K. This lead to truncation of the MongoDB size by approx. 25% of all records in the benchmark.

Performance Evaluations – Next Steps

- Initial tests confirm that Postgres' can handle many NoSQL workloads
- EDB is making the test scripts publicly available
- EDB encourages community participation to better define where Postgres should be used and where specialty solutions are appropriate
- Download the source at https://github.com/EnterpriseDB/pg_nosql_benchmark
- Join us to discuss the findings at http://bit.ly/EDB-NoSQL-Postgres-Benchmark

PG XDK

- Postgres Extended Document Type Developer Kit
- Provides end-to-end Web 2.0 example
- Deployed as free AMI
- First Version
  - Postgres 9.4 (beta) w. HSTORE and JSONB
  - Python, Django, Bootstrap, psycopg2 and nginx
- Next Version:
  PL/V8 & Node.js
- Final Version:
  Ruby on Rails



AWS AMI PG XDK v0.2 - ami-1616b57e

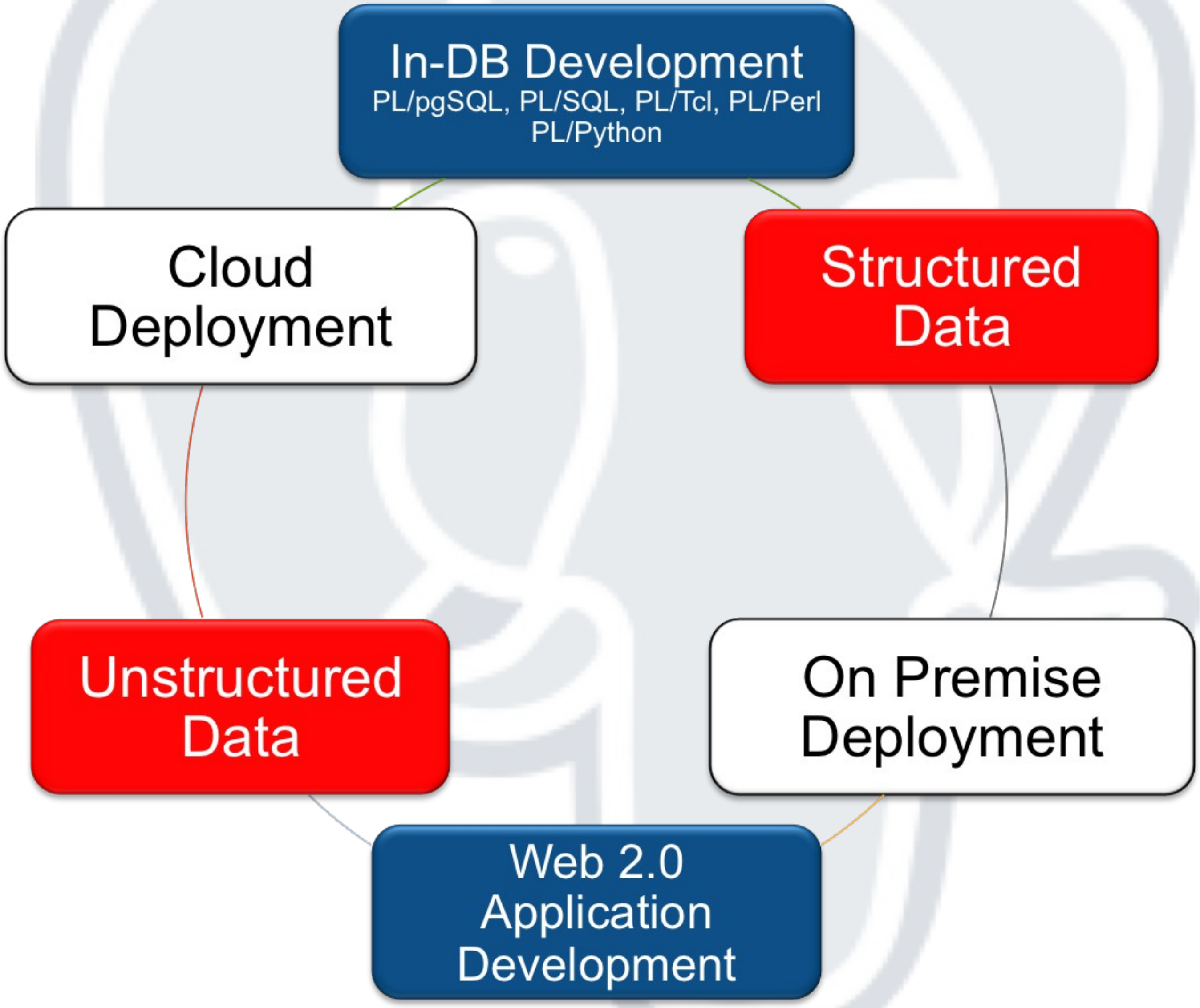
Installing PG XDK

- Select PG XDK v0.2 - ami-1616b57e on the AWS Console
- Use https://console.aws.amazon.com/ec2/v2/home?region=us-east-1#LaunchInstanceWizard:ami=ami-1616b57e
- Works with t2.micro (AWS Free Tier)
- Remember to enable HHTP access in the AWS console

Structured or Unstructured? “No SQL Only” or “Not Only SQL”?

- Structures and standards emerge!
- Data has references (products link to catalogues; products have bills of material; components appear in multiple products; storage locations link to ISO country tables)
- When the database has duplicate data entries, then the application has to manage updates in multiple places – what happens when there is no ACID transactional model?

Ultimate Flexibility with Postgres



Say yes to 'Not only SQL'

- Postgres overcomes many of the standard objections "It can't be done with a conventional database system"
- Postgres
  - Combines structured data and unstructured data (ANSI SQL and JSON/HSTORE)
  - Is faster (for many workloads) than than the leading NoSQL-only solution
  - Integrates easily with Web 2.0 application development environments
  - Can be deployed on-premise or in the cloud

Do more with Postgres – the Enterprise NoSQL Solution

Useful Resources

- Whitepapers @ **http://www.enterprisedb.com/nosql-for-enterprise**
  - PostgreSQL Advances to Meet NoSQL Challenges (business oriented)
  - Using the NoSQL Capabilities in Postgres (full of code examples)
- Run the NoSQL benchmark
  - **https://github.com/EnterpriseDB/pg_nosql_benchmark**
- Test drive PG XDK
- Check out the jsonbx repo: **https://github.com/erthalion/jsonbx**
  - JSON-modifying operators and functions (hopefully coming to PostgreSQL 9.5)

Do More With Postgres!

Flexible schemas: Faster development cycles

ENTERPRISEDB

Less complexity in your data environment

Document, key–value, and relational in one database

Data Integrity without silos