Using the NoSQL Capabilities in Postgres

An EnterpriseDB White Paper

For DBAs, Developers & Database Architects
July 2015

Table of Contents

Why NoSQL?

Businesses and technology teams are demanding greater flexibility, faster time to market and continuous refinement of existing systems. This requires an unprecedented agility in data management that traditional relational database management systems have been striving to deliver. NoSQL-only solutions, such as document stores and key-value stores, emerged to support incremental development methodologies where data models emerge as the application goes through cycles of agile development. This is in contrast to traditional methods of carefully crafting data models upfront using ER-diagramming, normal form analysis, and conceptual/logical/physical design frameworks.

PostgreSQL (often called Postgres) introduced JSON and HSTORE to provide solution architects and developers a schema-less data management option that is fully integrated with Postgres' robust ACID (Atomic, Consistent, Isolation and Durable) model. HSTORE has been an integral part of Postgres since 2006; JSON was first introduced in Postgres v9.2. With the upcoming release of Postgres v9.4 this fall, Postgres' NoSQL capabilities will dramatically expand and performance will skyrocket.

This paper reviews and illustrates Postgres' NoSQL capabilities in the context of Postgres' robust relational competencies. The paper also describes performance tests that demonstrate that Postgres, the leading open source relational database solution, is a superior platform for handling many NoSQL workloads.

Postgres NoSQL Capabilities

Postgres was originally architected to be an object-relational database designed specifically to enable extensibility. It supports objects, classes, custom data types and methods. In the early years of the Postgres project this was problematic as it slowed down development cycles because new code had to be fully integrated so everything would work with everything else. However, as Postgres has become more feature rich over the past 15 years, that original design hurdle has turned into a unique advantage. The fact that Postgres is an object-relational database means new capabilities can be developed and plugged into the database as needs evolve.

Using this level of extensibility, Postgres developers have expanded the database to include new features and capabilities as new workloads requiring greater flexibility in the data model emerged. The most relevant examples in the NoSQL discussion are JSON and HSTORE. With JSON and HSTORE, Postgres can support applications that require a great deal of flexibility in the data model.

Document Database – JSON

Document database capabilities in Postgres advanced significantly when support for the JSON data type was introduced in 2012 as part of Postgres 9.2. JSON (JavaScript Object Notation) is one of the most popular data-interchange formats on the web. It is supported by virtually every programming language in use today, and continues to gain traction. Some NoSQL-only systems, such as MongoDB, use JSON (or its more limited binary cousin BSON) as their native data interchange format.

Postgres offers robust support for JSON. Postgres has a JSON data type, which validates and stores JSON data and provides functions for extracting elements from JSON values. And, it offers the ability to easily encode query result sets using JSON. This last piece of functionality is particularly important, as it means that applications that prefer to work natively with JSON can easily obtain their data from Postgres in JSON.

Below are some examples of using JSON data in Postgres.

JSON data 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 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”},
      ]
}
```

Figure 1: JSON Data Examples

Extracting a list of products from 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
 AC3 Series Charger
 Phone Extended Warranty
 Phone Service Core Plan
(11 rows)
```

Figure 2: JSON Query Example

Extracting a list of products for the brand 'ACME'

```
SELECT DISTINCT
      data->>'name' AS "Product Name",
      data->>'price' AS "price"
FROM json_data
WHERE data->>'brand' = 'ACME';

 Product Name | price
--------------+-------
 AC3 Phone    | 200
 AC7 Phone    | 320
(2 rows)
```

Figure 3: JSON Query Example

In addition to the native JSON data type, Postgres v9.3, released in 2013, added a JSON parser and a variety of JSON functions. This means web application developers don't need translation layers in the code between the database and the web framework that uses JSON. JSON-formatted data can be sent directly to the database where Postgres will not only store the data, but properly validate it as well. With JSON functions, Postgres can read relational data from a table and return it to the application as valid JSON formatted strings. And, the relational data can be returned as JSON for either a single value or an entire record, as given below:

Fetching JSON data using SELECT

```
SELECT data FROM json_data;

data
-----------------------------------------------------
----------
 {"name": "Apple Phone", "type": "phone", "brand":
"ACME", "price": 200, "available": true,
"warranty_years": 1}
(1 row)
```

Figure 4: JSON Query Example

Extract the price of an Apple phone

```
SELECT
      data->>'price' as iphone_price
FROM  json_data
WHERE data->>'name'='Apple Phone';

 iphone_price
--------------
 200
(1 row)
```

Figure 5: JSON Query Example

Key-Value Pairs – HSTORE

The HSTORE contrib module, which can store key/value pairs within a single column, enables users to create a schema-less key-value store. But unlike with NoSQL-only solutions, a key-value store created in Postgres is ACID compliant.

HSTORE was introduced in Postgres 8.2 in 2006 and pre-dates many NoSQL advances. Its popularity has expanded in recent years with new demands for working with semi-structured data. It is a particularly handy tool for web developers or someone building an application that requires the ACID properties of Postgres as well as NoSQL capabilities.

HSTORE is not hierarchical, but the HSTORE data type offered advanced indexing support early on, which made it the solution of choice for many applications. It is particularly useful for sparse attributes – instead of creating a large number of columns, each of which will be non-null for only a small percentage of the records in the table, database administrators can create a single HSTORE column and include, for each row, only those keys which pertain to that record. For instance, this is quite useful for storing multiple product descriptions in a single table where each product only shares a few attributes like name, price and weight, but have many different attributes based on the type of product.

Just like JSON, HSTORE can be used to emulate a schema-less database when that is desirable. Ultimately, this fills a unique need in relational table storage by not requiring additional space for attributes that will never have a value for many records. It allows database administrators to store very different types of records with different attributes in the same table yet easily query them using SQL.

Integrating JSON and HSTORE

There are also functions that convert Postgres-maintained key-value data to JSON formatted data, which increases the flexibility and scope of NoSQL-like applications that can be addressed by Postgres.

Following are some examples:

```
HSTORE and JSON Integration 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 ;

data
------------------------------------------------------
 "cost"=>"500", "product"=>"iphone",
"provider"=>"Apple"
(1 row)
Convert HSTORE data to JSON
SELECT hstore_to_json(data) FROM hstore_data ;

hstore_to_json
--------------------------------------------------------
-------
 {"cost": "500", "product": "iphone", "provider":
"Apple"}
(1 row)


SELECT hstore_to_json(data)->>'cost' as price FROM
hstore_data ;
 price
-------
 500
```

Figure 6: Integration JSON and HSTORE

In summary, the JSON data, operator and function enhancements mean that developing NoSQL applications has become much easier.

JSONB – Binary JSON

Postgres 9.4 introduces JSONB, a second JSON type with a binary storage format. There are some significant differences between JSONB in Postgres and BSON, which is used by one of the largest document-only database providers. JSONB uses an internal storage format that is not exposed to clients; JSONB values are sent and received using the JSON text representation. BSON stands for Binary JSON, but in fact not all JSON values can be represented using BSON. For example, BSON cannot represent an integer or floating-point number with more than 64 bits of precision, whereas JSONB can represent arbitrary JSON values. Users of BSON-based solutions should be aware of this limitation to avoid data loss.

JSONB/BSON Precision Discussion

```
Insert a 64 bit number into BSON:

db.test.insert(
{
"precision": 1.0000000000000000000000000000000000002
      })

Search for a number greater than 0

db.test.find({precision:{$gt:1}},{precision:1})
Result: 0 rows

Same example in Postgres 9.4

SELECT * FROM json_data where data->>'precision' >
'1';
               data
----------------------------------------------------
----
 {"precision":
1.0000000000000000000000000000000000002}
(1 row)
```

Figure 7: BSON Challenges with High Precision Data

JSON and PL/V8 – Javascript for Postgres

PL/V8 Function Example

```
test=# CREATE EXTENSION plv8;
CREATE EXTENSION

CREATE OR REPLACE FUNCTION json_data_update(data json, field text,
value text)
RETURNS jsonb
LANGUAGE plv8 STABLE STRICT
     AS $$
       var data = data;
       var val  = value;
       data[field] = val;
      return JSON.stringify(data);
 $$;
```

This function takes three inputs:

- The name of the JSON data field,
- The tag
- The new value

Below is an example where the user is updating the value of 'price' for the AC3 phone, using the above function:

```
SELECT
      data as old_price_data,
      json_data_update
         (
            data,
            'price'::text,
            '200'::text
         ) as new_price_data
FROM
      json_data
WHERE data->>'name' = 'AC3 Phone';
```

Result

```
old_price_data | {"name": "AC3 Phone", "type": "phone", "brand":
"ACME", "price": 200, "available": true, "warranty_years": 1}
 new_price_data | "{\"name\": \"AC3
Phone\", \"type\": \"phone\", \"brand\": \"ACME\", \"price\":
200, \"available\": true, \"warranty_years\": 1}"
```

Postgres provides Javascript capabilities right in the database, which allows developers who know Javascript to write code inside the database using the same JavaScript engine that powers the web. V8 is a powerful and fast JavaScript engine that was developed by Google; in addition to powering Google Chrome it is also at the heart of Node.js. V8 was designed from the beginning to work on the client and on the server. V8 is available in Postgres as PL/V8, an add-on for Postgres. **Figure 8: PL/V8 Examples**

Combining ANSI SQL Queries and JSON Queries

One of Postgres' key strengths is the easy integration of conventional SQL statements, for ANSI SQL tables and records, with JSON and HSTORE references pointing documents and key-value pairs. Because JSON and HSTORE are extensions of the underlying Postgres model, the queries use the same syntax, run in the same ACID transactional environment, and rely on the same query planner, optimizer and indexing technologies as conventional SQL-only queries.

Examples of Integrating ANSI SQL and JSON queries

Find a product that has associated warranty information and select the price from the products table (Conventional SQL statements are in *italics* and NoSQL references are **bold**).

```
SELECT DISTINCT
      product_type,
      data->>'type' as service,
      data->>'price' as "price(USD)"FROM
      json_data
JOIN productsON (products.service_type = json_data.data->>'type')
WHERE (data->>'warranty_years') > 0;
      product_type         |          service            | price(USD)
-------------------------+-----------------------------+------------
 AC3 Case Black          | [ "accessory", "case" ]     | 12
 AC3 Case Black          | [ "accessory", "case" ]     | 12.5
 AC3 Case Green          | [ "accessory", "case" ]     | 12
 AC3 Case Green          | [ "accessory", "case" ]     | 12.5
 AC3 Case Red            | [ "accessory", "case" ]     | 12
 AC3 Case Red            | [ "accessory", "case" ]     | 12.5
 AC3 Phone               | phone                       | 200
 AC3 Phone               | phone                       | 320
 AC3 Series Charger      | [ "accessory", "charger" ]  | 19
 AC7 Phone               | phone                       | 200
 AC7 Phone               | phone                       | 320
 Phone Extended Warranty | warranty                    | 38(12 rows)
```

Find the product_type and brand that is in stock (Conventional SQL statements are in *italics* and NoSQL references are **bold**).

```
SELECT DISTINCT
      product_type, data->>'brand' as Brand,
      data->>'available' as Availability
FROM json_data
JOIN productsON (products.product_type=json_data.data->>'name')WHERE
json_data.data->>'available'=true; product_type | brand |
availability--------------+-------+--------------
 AC3 Phone    | ACME   | true
 AC3 Case Red |        | true
(2 rows)
```

Figure 9: Postgres Queries Combining JSON and ANSI SQL

Bridging Between ANSI SQL and JSON

Postgres provides a number of functions to bridge between JSON and ANSI SQL. This is an important capability when applications and data models mature, and designers start to recognize emerging data structures and relationships.

Postgres can create a bridge between ANSI SQL and JSON, for example by making a ANSI SQL table look like a JSON data set. This capability allows developers and DBAs to start with an unstructured data set, and as the project progresses, adjust the balance between structured and unstructured data.

Bridging between ANSI 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

(3 rows)

Select query returning the same result as a JSON data
set

SELECT ROW_TO_JSON(products) FROM products;
            row_to_json
-----------------------------------
 {"id":1,"product_name":"iPhone"}
 {"id":2,"product_name":"Samsung"}
 {"id":3,"product_name":"Nokia"}

(3 rows)
```

Figure 10: Postgres Queries Combining JSON and ANSI SQL

EDB Comparative Performance Tests

EDB has started to conduct comparative evaluations to help users correctly assess Postgres' NoSQL capabilities.

The initial set of tests compared MongoDB v2.6 to Postgres v9.4 beta, on single instances. Both systems were installed on Amazon Web Services M3.2XLARGE instances with 32GB of memory.

MongoDB and Postgres were installed 'out of the box'; neither database was manually tuned for this workload.

The tests included:

- A machine-generated set of 50 million JSON documents, similar to the first JSON example in Figure 1 (enhanced with one large 'description:' field that was populated in 60% of the documents using random text)
- A load of the data into MongoDB (using IMPORT) and Postgres (using COPY)
- 50 million individual insert operations for the same data
- Multiple select statements for random records with both databases returning all records in the query

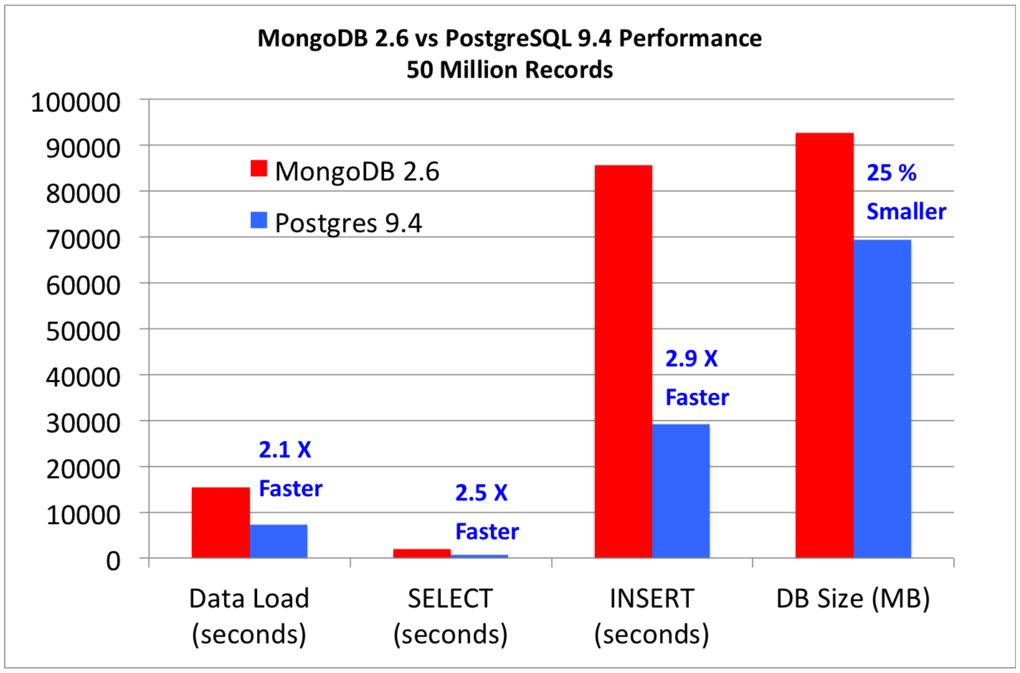
Table 1 summarizes the results of our findings:

- Ingestion of high volumes of data was approximately 2.1 times faster in Postgres
- MongoDB consumed 33% more the disk space
- Data inserts took almost 3 times longer in MongoDB
- Data selection took more than 2.5 times longer in MongoDB than in Postgres

Table 1: MongoDB and Postgres Comparison (Absolute)

	MongoDB 2.6	**PostgreSQL 9.4**
Data load (s)	15391	7319
Inserts (s)	85639	29125
Selects (s)	1929	753
DB Size (GB)	92.63	69.36

Figure 10: Relative Performance Comparison of MongoDB 2.6 with PostgreSQL 9.4



Postgres Plus Advanced Server includes a procedural language called EnterpriseDB SPL that closely matches Oracle's PL/SQL procedural language. Like PL/SQL, SPL is a highly productive, block-structured procedural programming language for writing custom procedures, functions, and triggers. The close similarity between EnterpriseDB's SPL and Oracle's PL/SQL also enables Postgres Plus Advanced Server to support Oracle-style packages of procedures, functions and variables.

Postgres: NoSQL for the Enterprise

SQL databases like Postgres have added features to fill the gap that motivated the rise and development of NoSQL-only technologies, and will continue to provide capabilities that NoSQL-only technologies simply cannot. Users of NoSQL-only technologies are finding they still need relational and transactional capabilities when working with unstructured data and seeking ways to combine data within a single, enterprise-grade environment.

The ability of Postgres to support key-value stores and documents within the same database empowers users to address expanding demands using proven, best-in-class open source technologies. Utilizing NoSQL capabilities within Postgres to address more data problems instead of turning immediately to a niche NoSQL solution ultimately means lower costs, less risk and less complexity while delivering enterprise-class workloads with ACID compliance and ensuring the long-term viability of enterprise data.

Long-standing capabilities that have new uses and continuing advances will enable Postgres to play a significant role in the data center and in the cloud long into the future, even as new data challenges emerge.

Get Started Today. Let EnterpriseDB help you build and execute your game plan. Contact us at +1-877-377-4352 or +1-781-357-3390, or send an email to sales@enterprisedb.com to get started today on your path to database independence.

About EnterpriseDB

EntepriseDB is the only worldwide provider of enterprise-class products and services based on PostgreSQL, the worlds most advanced and independent open source database.

Postgres Plus Advanced Server provides the most popular enterprise class features found in the leading proprietary products but at a dramatically lower total cost of ownership across transaction intensive as well as read intensive applications. Advanced Server also enables seamless migrations from Oracle® that save up to 90% of the cost of typical Migrations.

EnterpriseDB employs a number of industry thought leaders and more PostgreSQL open source community experts than any other organization including core team members, major contributors Committers. Our expertise allows us to provide customers in all market segments unparalleled 24x7 support, value packed software subscriptions, consulting engagements and targeted training services.

For more information, please visit http://www.enterprisedb.com/.

20140815