Improving performance of schemaless document storage in PostgreSQL using BSON

Geoffrey Litt Seth Thompson John Whittaker
Yale University
geoffrey.litt@yale.edu seth.thompson@yale.edu john.whittaker@yale.edu

Abstract

NoSQL database systems have recently been gaining in popularity. These databases provide schema flexibility compared to traditional relational databases like PostgreSQL, but consequently give up certain desirable features such as ACID guarantees. One compromise is to store schemaless documents within a relational database—an architecture which PostgreSQL has recently made possible by adding native support for JSON (JavaScript Object Notation), a common format for multi-level key-value documents.

Currently, PostgreSQL provides validation of JSON documents upon input and native operators for querying operators within a JSON document. However, it stores JSON documents internally as text, which is inefficient for many use cases. In this paper, we implement support for BSON (Binary JSON), a lightweight binary encoding format for JSON-like documents which enables fast traversals. We observe that using BSON to store documents increases performance by two to eight times when querying keys within documents, without compromising document insertion times.

General Terms NoSQL, RDBMS

Keywords PostgreSQL, JSON, BSON

1. Introduction

With the rapid growth over the past decade of big data fueled by data-driven applications, the field of database systems has exploded, in terms of both size and variety. The traditional Relational Database Management Systems (RDBMSs) that have ruled the marketplace for decades are still the best option for many use cases, but NoSQL systems such as MongoDB [1] are increasing in popularity. Some benefits provided by NoSQL databases include a simpler key-value data model, strong support for cluster environments (especially those composed of commodity hardware), and emphasis on availability over consistency.

Some have suggested that a way for RDBMSs to compete with NoSQL competitors—or at least to innovate in a time of changing needs—is to offer the ability to store schemaless documents within a traditional relational framework [2]. This feature allows users to leverage the ACID guarantees, performance benefits, code maturity, and other advantages of a relational database, while also offering schema flexibility when required.

The open-source database PostgreSQL [3] recently released such support. In late 2012, the latest stable build, version 9.2, added a native JSON data type. JSON is a human-readable format for schemaless document storage. Postgres' implementation simply applied JSON validation to a stored text field containing a string representing each document. At the time of the release, querying values within a document required using an external JavaScript engine for parsing sub-document elements. The current development build (9.3dev) has improved the built-in functionality by adding native operators to query values within a JSON document, without the need for shared libraries or external functions.

The operator syntax is simple: an operation like `SELECT data->'key1' FROM json_documents` will return the value associated with the key `"key1"`, in the JSON-type data column "data".

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CPSC 438 Final Project April 29, 2013, New Haven, CT.
Copyright © 2013 . . . $15.00

Although these developments are clearly useful, their scope is limited by the fact that the JSON type in PostgreSQL remains a simple syntax validation. JSON documents are stored in human-readable text format in the database—without any optimizations for efficient storage or fast traversal. This approach enables documents to be easily passed to clients and end users, but does not necessarily enable performant querying of values within a document.

In contrast, the popular NoSQL database MongoDB stores documents internally in the BSON binary format. BSON is a lightweight binary encoding for JSON-like documents, which is specifically designed for speedy traversal compared to the human-readable JSON format.

To clarify a matter of semantics, this paper uses "JSON-like" to refer specifically to the format of the hierarchical data document shared by both JSON proper and BSON. This format is identical to the end user (human or machine) in that both JSON proper and BSON documents are simply strings of keys and values organized into dictionaries or arrays by braces and brackets, respectively. In the case of JSON proper, this format is also how the document is encoded for storage. But in the case of BSON, the interchange format is further encoded for storage into a binary blob, a process which is described in more detail in Section 2.3.

In this paper, we describe our implementation of a native BSON type in PostgreSQL. We implement a user-facing interface which is nearly identical to the JSON interface—users insert documents using JSON syntax (which enables nesting of objects, arrays, etc.), and query values within a document using the operator syntax introduced by PostgreSQL 9.3. However, the backend storage mechanism is completely different. Our implementation parses the JSON document into key-value pairs upon insertion, and then constructs a BSON document which encodes the same information in BSON format. Upon querying, instead of having to parse the text document as in JSON, the BSON implementation can directly iterate over keys, providing significant performance advantages.

We found performance gains across data sets of widely varying tuple size and document size. In some tests, queries performed on keys in BSON documents ran over eight times as fast as those on corresponding JSON documents. BSON was performant for many data types and on deep queries of large numbers of tuples, suggesting that our results hold for a wide variety of data sets. Ultimately, we believe that BSON's advantages merit its replacement of JSON in almost all use cases.

The rest of the paper is organized as follows: Section 2 provides background information on PostgreSQL, JSON, and BSON. Section 3 describes our implementation of BSON in PostgreSQL. Section 4 presents performance results when querying values inside a document in JSON and BSON, with a variety of data sets and use cases. Section 5 gives our conclusions and plans for future work.

2. Background

2.1 PostgreSQL

PostgreSQL [3] is an open-source relational database system. It implements the majority of the most recent ISO and ANSI standard for the SQL database query language, has ACID-guarantees, and is fully transactional. Among PostgreSQL's advantages are its availability for numerous platforms, highly extensible architecture, support for complex data types, and mature, 15-year-old codebase. We decided to modify PostgreSQL because it is a reliable and efficient database system, and because it already had support for schemaless document storage.

PostgreSQL currently has three methods for storing schemaless data: XML, hstore, and JSON. XML is a hierarchical structured data format which became a W3C standard in 1998. The format was the backbone of the SOAP network message passing protocol and is still used today by many Java applications as a markup language. However, XML's age is starting to show: many criticize its verbosity and complex feature set [4]. It is also the slowest of PostgreSQL's document storage types, and features no indexing [5].

The second unstructured storage type implemented in PostgreSQL is hstore, which maps keys to string values or other hstore values. Although hstore is highly expressive, the lack of nested documents and multiple types limits its functionality as a versatile schemaless store.

The last, and most recently added schema, is JSON, described in detail in the following section. Initially conceived as JavaScript's structure declaration format, JSON was turned into a language-agnostic protocol. Today it enjoys ubiquity as a data exchange format

```
{
 "firstName": "George",
 "lastName": "Washington,
 "phoneNumbers": [
   {
     "type": "home",
     "number": "203 123 4567"
   },
   {
     "type": "business",
     "number: "202 345 6789"
   }
 ]
}
```

Figure 1. An example JSON document

across many modern website. JSON is the most performant of PostgreSQL's document storage methods [5]

2.2 JSON

JavaScript Object Notation [6] is a lightweight data-interchange format derived from JavaScript. It is designed to be easy for humans to read and write and for machines to parse. An object in JSON is a set of key-value pairs which can be organized into arrays and nested objects. Values can be strings, booleans, or arbitrary-precision numbers. This allows for extremely flexible object representation, including embedding of an arbitrary number of related objects, making JSON a suitable data format for a NoSQL database. Figure 1 is an example of JSON (and any equivalent "JSON-like" format).

This example demonstrates the power of JSON as a representation format—the document can store any number of phone numbers for the user, without requiring another table and a join operation.

2.3 BSON

BSON [7] is a binary serialization of JSON-like documents, developed for the NoSQL database MongoDB and designed for performant traversal. It is generally capable of storing any JSON document, with some minor caveats. The biggest difference is that JSON stores numbers as arbitrary precision character strings, whereas BSON can only store numbers as 32- and 64-bit integers or 64-bit double precision floats, in order to make storage simpler. (Of course, a careful programmer can easily store arbitrary precision numbers as strings in BSON, just as in JSON.) Additionally, BSON enforces a maximum document size of 16MB (which is unlikely to be exceeded in most circumstances).

BSON has several advantages compared to JSON which enable fast traversal. Many types (e.g. integers, booleans) are stored as fixed length fields, as opposed to variable length fields which must be parsed in JSON. For example, a boolean in BSON is represented by a single byte, as opposed to several bytes representing the word "true" or "false" in JSON. Additionally, BSON uses native C data types for many types such as integers and floats, which enables compact storage and more efficient parsing. Finally, BSON also encodes length information for variable length fields like strings, so they can be skipped over easily when querying for a specific key deep in the document. BSON is not necessarily more compact on disk than the equivalent JSON, but can almost always be traversed more quickly for these reasons.

3. Implementing BSON in PostgreSQL

We heavily utilized two open-source libraries to complete this project. To implement BSON writing and parsing functionality, we used the `mongo-c-driver` library [8] from MongoDB. We also used the `json-c` library [9] to implement JSON parsing, which was necessary to translate user JSON input to a series of key-value pairs which could be encoded as BSON.

The implementation can be divided into two main parts: storage of input, and querying keys. When storing input, we use the JSON library to parse the input, and iteratively construct a BSON object from the parsed object. For querying within the BSON document, we created functions which use the BSON library to find the desired key, and registered operators in the PostgreSQL function catalog to execute these functions on BSON data.

The operators we created to query BSON data are a superset of the operators which can be used to query JSON in PostgreSQL. Because BSON stores data as native C data types, we added two operators, `-&>` and `-#>`, which take advantage of this feature to return boolean and integer native data types instead of strings. Whereas acquiring an integer or boolean type from a JSON document requires an explicit cast of the queried data, our BSON operators can directly return these native types with no casting needed.

```
{
  "user":{
    "name":"Karen",
    "lastname":"Smith",
    "age":15,
    "country":"Norway",
    "birth_year":1952,
    "birth_month":11,
    "birth_day":7,
    "phones":[4968387550,4408548944],
    "address":"86 Main St., Anytown, USA",
    "registered":false,
    "new":false
  },
  "friends":[
     {"name": "Bob",...},
     {"name": "Sally",...},
     ...
   ]
}
```

Figure 2. Example document to simulate a user record in our performance evaluation

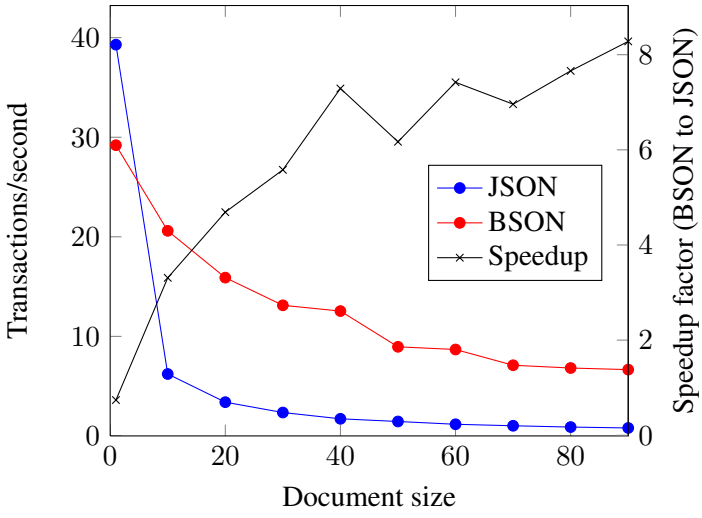

Figure 3. Relative performance of BSON and JSON for increasing document size over a constant number of tuples

Also, because we added the BSON type alongside the JSON type, as opposed to modifying the JSON type to be internally represented as BSON, we were able to easily test the two data types together in the same database.

4. Evaluation

4.1 User document testing

To achieve realistic results, most of our tests used documents representing information for a user of a social network application, including an array of embedded information about other users who are ”friends” with a given user. We randomly generated documents similar to that in Figure 2.

4.1.1 Varying document size

Our first performance test examined performance when querying a document key, while varying document size. For each trial, we generated a table with a single column of type JSON or BSON, and inserted 1000 rows into these tables, each row containing one user document of the type above. We then executed a query on a key in the document (e.g. `SELECT data -> 'friends' -> 50 -> 'registered' FROM json`) on each entire table, using the `pgbench` tool built into PostgreSQL to repeat the query 10 times and determine an average transactions per second metric for both JSON and BSON.

Then, to determine the effect of document size on the relative performance of JSON and BSON querying, between trials we varied the number of embedded ”friend” user documents from 1 to 90. The results of this test are shown in Figure 3, along with a speedup ratio between BSON and JSON.

BSON performs much faster than JSON for most document sizes, and the speedup ratio increases with larger document size. With 90 embedded friends, BSON performs the select query 8.28 times faster than JSON.

This result serves as preliminary evidence that BSON can traverse a document much more rapidly than JSON, leading to higher performance. Our proposed explanation is that as document size increases, time spent traversing the document searching for a given key takes a greater fraction of total query execution time (compared to other overhead), and the traversal information in the BSON encoding becomes increasingly helpful for performance. However, because other factors are also changing (e.g. total data size increases with document size), further exploration is necessary to establish that the traversal speed is responsible for the speedup, as opposed to other factors like BSON’s more compact representation on disk.

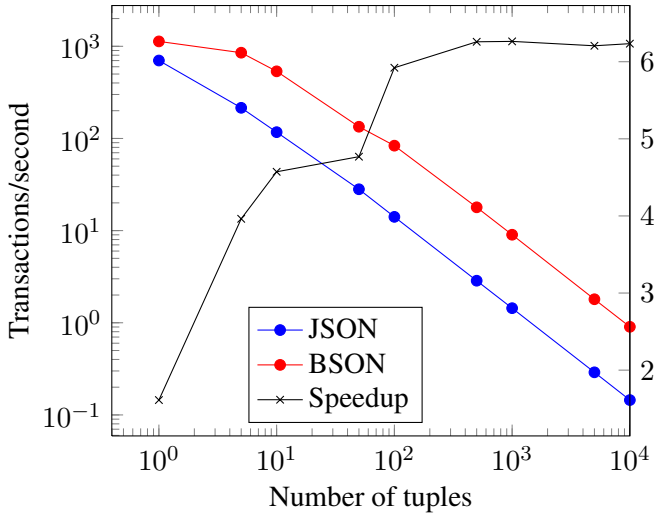

Figure 4. Relative performance of BSON and JSON for increasing number tuples with a constant document size

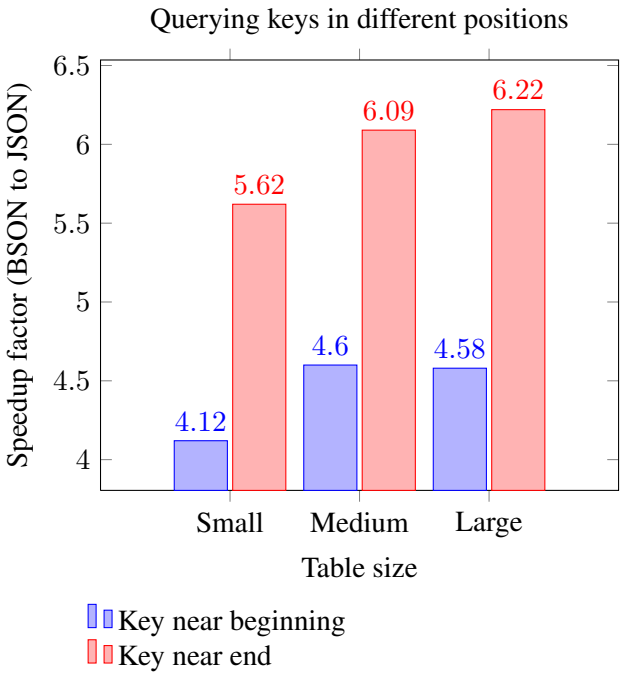

Figure 5. Comparison of the BSON speedup factor for keys near the beginning and end of documents.

4.1.2 Varying table size

In our next test, we explored again used "user" documents, but this time kept document size constant at 50 embedded friends. We incrementally increased the number of rows in the table from 1 to 10,000, and observed the performance of JSON and BSON on a `SELECT` query on a key in the document, over the entire table. The results are shown in Figure 4 (notice the logarithmic scaling).

After the number of tuples exceeded 100, the BSON speedup factor remained constant at about 6 times JSON's query speed. This shows that the BSON datatype can efficiently maintain its performance advantage for relatively large table sizes.

4.1.3 Key position effect

We devised a test to conclusively determine whether BSON's traversal speed is responsible for its faster performance: comparing performance when querying a key near the beginning or the end of a document.

We queried user documents with 50 embedded friends for this test. For the key near the beginning of the document, we queried the "name" key which is first in the user document. For the key near the end, we queried a key within one of the user's friends stored near the end of the document. We tried the experiment with several table sizes for variety.

Figure 5 shows the results of these tests, comparing the BSON speedup factor for each of the key positions across three table sizes. Clearly, the BSON speedup factor is significantly greater when querying a key near the end of a document.

This result indicates that the BSON implementation is gaining performance benefits from traversal speed. Both the Postgres JSON implementation and our BSON implementation iterate through a document one key at a time, searching for a match. The JSON parser has to go character by character parsing the document, whereas the BSON parser can easily skip over all values using included traversal information. As previously mentioned, the BSON format is specifically designed for easy traversal, and this test shows off the performance advantages of that design.

4.2 Document composition testing

Next, we tested the performance of JSON and BSON on documents storing only values of one specific type of data: strings, integers, and booleans. These documents do not take advantage of the flexibility of a schemaless database, but because BSON has efficient storage mechanisms for integer and boolean types, they can further demonstrate some of the primary advantages of the format.

We created JSON and BSON tables with 10,000 documents, where each document was made of 1,000

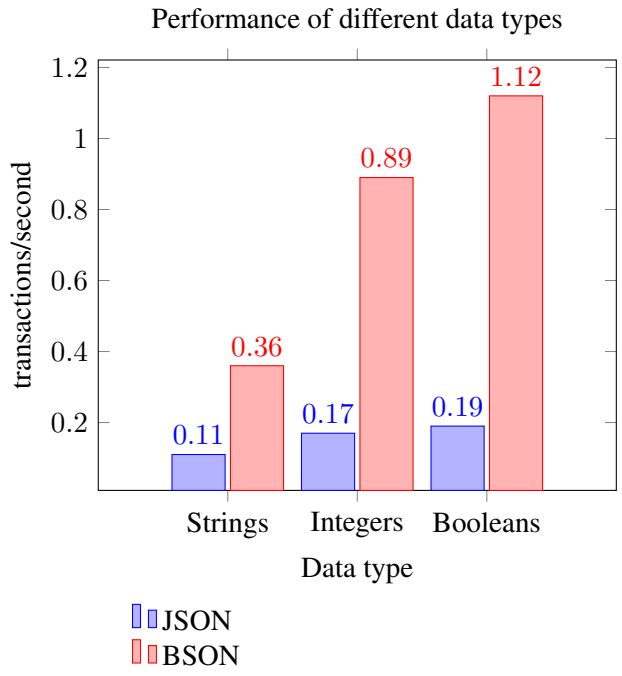

Figure 6. Comparison of the BSON speedup factor for documents containing exclusively one data type.

key-value pairs. All the values in each document were either booleans, 32-bit integers, or 10-character-long strings (all randomly generated), depending on the type being tested. We then queried a key in the middle of the document across all rows for both JSON and BSON, measuring performance of the queries. The transaction speeds in JSON and BSON documents for each data type are shown in Figure 6.

As expected, BSON provides a speedup of over five times for documents containing integer and boolean types. BSON stores integers in a fixed-width 4-byte binary format, and stores booleans as a single byte; in contrast, JSON stores integers as strings of digits (32 bit integers can take up to 10 characters to represent), and booleans as several characters representing the word ”true” or ”false”. As a result, BSON documents with integers and booleans are much smaller on disk than the corresponding JSON documents, reducing disk I/O time. The traversal benefits of the BSON parser which have been discussed also apply. This result suggests that applications which store documents with many integers, booleans, and other fixed length types would generally benefit most from using the BSON format, due to the storage compression benefits.

	1,000 tuples	10,000 tuples	100,000 tuples
JSON	6.100s	62.09s	630000s
BSON	6.139s	62.95s	635600s

Figure 7. Comparison of total loading times between JSON and BSON for 1000, 10,000, & 100,000 inserts.

Documents containing strings also gain a performance advantage of over three times from using BSON, because BSON encodes length information before a variable length string, making traversal easier. However, string data still takes up the same number of bytes in JSON and BSON, so unlike in the integer and boolean cases, BSON gains no disk I/O advantage.

This set of tests demonstrates that BSON has two different advantages which combine to provide a performance boost: it takes less computation time to traverse because it uses encoded metadata to skip by element instead of searching for keys character by character. Moreover, it takes less space on disk for certain kinds of data (but not all), often reducing I/O time.

4.3 Loading speed

PostgreSQL’s native JSON type directly stores the user’s input string to disk, whereas our BSON type parses the input into a more efficient format upon insertion. Therefore, although BSON has clearly been shown to have performance benefits upon read, insertion performance is a legitimate concern.

However, we measured write times and discovered that BSON loads almost exactly as fast as JSON. In fact, insertion of 1000, 10,000, & 100,000 tuples into a table of BSON documents was less than one percent slower than the same insertions into a column of JSON documents. Figure 7 details the loading times in seconds. Clearly, BSON’s performance gain in subdocument querying does not come at the expense of performant loading times.

In fact, PostgreSQL’s native JSON type also does parsing upon input, to validate the JSON string being inserted, so BSON is not at a disadvantage because of parsing upon insertion. We currently still use the built-in JSON validator to validate user input, and if we developed the code further, we could use the BSON parser to validate input and return appropriate errors, eliminating the need for the JSON validator and possibly making BSON inserts even faster than JSON inserts.

5. Conclusions and Future Work

This paper has presented an implementation of BSON for PostgreSQL which provides significant performance benefits over text-based JSON when querying within a schemaless document. As RDBMSs try to compete with NoSQL databases by offering support for storage of documents, read performance will be a crucial differentiator, and implementing performant data storage formats such as BSON could prove to be a major equalizer.

In future work, we plan to extend our code to cover all data types, and add further operators for efficient querying. Additionally, our BSON implementation is currently baked into a fork of the PostgreSQL source code, but it would be far more useful to the community if it were implemented as an external module. We plan to do further development work to make the code production-ready, add unit tests, and release it as an open source extension.

References

[1] MongoDB. *www.mongodb.org*.

[2] A. Floratou, N. Teletia, D. DeWitt, J. Patel, and D. Zhang. Can the Elephants Handle the NoSQL Onslaught? In *Proceedings of the VLDB Endowment*, Vol. 5, No. 12.

[3] PostgreSQL. *www.postgresql.com*.

[4] J. Atwood. XML: The Angle Bracket Tax. *http://www.codinghorror.com/blog/2008/05/xml-the-angle-bracket-tax.html*.

[5] C. Pettus. PostgreSQL as a Schemaless Database. *http://nosql.mypopescu.com/post/47692111874/posgresql-as-a-schemaless-database*.

[6] JSON. *json.org*.

[7] BSON. *bsonspec.org*.

[8] mongo-c-driver. *github.com/mongodb/mongo-c-driver*

[9] json-c. *github.com/json-c/json-c*