Comparing JSON-based databases to relational systems when undertaking complex queries on intricately structured data

Jeff Wintersinger
CSC 2508
January 5, 2015

Abstract

JavaScript Object Notation (JSON) is increasingly used not only as a data exchange format, but as a data storage format as well. With this increasing use comes a desire by users to execute complex online-analytical-processing queries against intricately structured documents without first converting them to a normalized relational format. This study explores whether JSON-based systems offer a sufficient richness to express complex queries against JSON documents with intricate structure, and the performance implications of doing so. It examines several systems, including PostgreSQL 9.4's binary JSON format, MongoDB, and, as a baseline, a traditional normalized relational PostgreSQL-based schema. It finds that JSON-based systems are, in the aggregate, competitive relative to their relational peers, but that performance differs dramatically from query to query, varying by orders of magnitude between systems.

Background

JavaScript Object Notation (JSON) (1) is increasingly used as a data storage format. Relative to traditional relational storage systems, JSON offers two primary advantages. Firstly, it avoids the object-relational impedance mismatch stemming from using a relational storage system from an object-oriented language (2). Thus, with JSON able to represent complex documents with nested structures, it may be used as the data exchange format across the network, the input to database storage systems, and the output from the same. Secondly, JSON is schemaless, which obviates the need for up-front schema design, and allows the format of stored data to evolve over time. While this characteristic may be detrimental to projects requiring that all stored data adhere strictly to a common format, it is nevertheless beneficial in some domains.

The goal of this project was to determine whether several newly released systems can efficiently answer intricate online-analytical-processing (OLAP) queries against complex JSON documents. To do so, a large corpus of JSON documents representing movies, their casts, and user-written reviews was generated randomly (Code Listing 1), yet structured so as to satisfy several constraints concerning relationships between them. This process and the associated constraints is described at length in Methods. Documents were structured so as to mirror the nature of those that might be served by services such as The Movie Database (3) or Rotten Tomatoes (4), representing a use case in which denormalized documents storing a wealth of movie-related information are served to API consumers.

Next, six complex queries were defined, answering questions such as which movies had at least half their reviewers from Canada, and which movie pairs share at least one actor in common. These queries were translated into the JSON query language offered by the newly released PostgreSQL 9.4 (henceforth PostgreSQL-JSON), as well as the query language supported by MongoDB. For comparison to the traditional relational model, the JSON documents were also decomposed into their base relations and fully normalized so they could be stored in PostgreSQL's existing relational system (henceforth PostgreSQL-relational), with the queries then translated into standard relational ones.

MongoDB was chosen for inclusion in this study because it is the most prominent non-relational JSON document database. PostgreSQL-JSON was included as its approach is similar to that of the

commercial Oracle Database system, yet is freely available. (Though Oracle offers a free version of their database system for such uses (5), it does not support the JSON operations described in the afore-referenced paper.) PostgreSQL's JSON support was introduced in version 9.3, released in September 2013 (6), and substantially expanded in version 9.4, released in December 2014 (7). Notably, PostgreSQL 9.4 introduced the *jsonb* data type, a binary format offering more efficient querying, a host of new operators necessary to express the rich queries used by this study, and the ability to be indexed by PostgreSQL's generalized inverted indices. In this regard, PostgreSQL's approach satisfies the three principles adopted by the Oracle Database (5): PostgreSQL-JSON satisfies the storage principle, requiring that documents be stored in their native JSON formats rather than be shredded into relational documents, as with Argo (2); it satisfies the query principle, requiring that SQL be retained for querying after being extended with JSON-specific operators for querying and navigating JSON documents, rather than being replaced by an entirely new language; and it satisfies the index principle, requiring that JSON documents be indexed using an inverted index to support efficient querying. Consequently, PostgreSQL-JSON offers the same advantage as Oracle Database, by which JSON data is stored in existing relational systems rather than requiring replacement by entirely new models. As a baseline, PostgreSQL-relational was selected to demonstrate how performance on JSON documents compares to that of traditional normalized relational data stores. By using PostgreSQL for this task, with queries possessing similar structures to those used in PostgreSQL-JSON, a reasonable illustration of the penalty paid for using JSON is obtained, leaving constant other system variables.

Three other systems were considered for inclusion in this study but rejected. The first such was Argo, which decomposes JSON documents into relational data suitable for storage in traditional databases (2). The query operations offered by Argo were suitable only for simple online-transaction-processing (OLTP) queries, and lacked the expressive power required for the more intricate OLAP queries used here. This study also improves upon Argo by using more complex documents with more complicated queries, better reflecting the structure of documents to which JSON is most often applied. The second additional system considered was Oracle Database. As previously mentioned, Oracle does not provide a freely available version of their system with JSON support, inhibiting evaluation. Finally, attempts were made to integrate ToroDB (8), which, like Argo and PostgreSQL-JSON, stores JSON-formatted data in PostgreSQL. Notably, it supports the MongoDB wire protocol, meaning that existing MongoDB-using applications should be able to switch transparently to ToroDB. Unfortunately, ToroDB was insufficiently mature for this study, with four of the six MongoDB queries failing to execute against ToroDB. These issues may be remedied with the release of ToroDB 1.0, scheduled for Q1 2015 (8).

No previous studies have compared the performance of complex queries on intricate JSON documents across multiple systems. While Argo (2) and Oracle (9) compared their systems relative to others, these studies used only straightforward queries against simple documents. Though other studies have undertaken exhaustive examinations of key-value stores (10), little work has been done with regard to JSON.

Methods

All source code used in this project is available at https://github.com/jwintersinger/csc2508-project/. Exact data sets used are available upon request; alternatively, new data sets may be generated using the generator.py script included with the code. Queries 1 through 6 used for MongoDB, PostgreSQL-JSON, and PostgreSQL-relational are available in the files mongo_queries.py, postgres_json_queries.py, and postgres_relational_queries.py, respectively. The insertion procedures used for the PostgreSQL systems are available in postgres_json_inserter.py and postgres_relational_inserter.py.

Data generation and insertion

Documents representing movies were generated using a Python script, with each record using the format shown in Code Listing 1. Each record included a unique integer ID, title, release date, list of one or more genres, list of actors (each possessing a unique integer ID, birth date, and name), and list of reviews (each possessing a unique integer ID, integer rating from 1 to 5, text description, and user record, comprised of name, country, and list of recent purchases). Movie titles, actor names, and review text descriptions were randomly generated alphanumeric strings. Movie genres were randomly drawn from a list of 22 pre-existing genres.

Movie release dates and actor birth dates were drawn from a uniform distribution composed of dates extending from the years 1900 to 2015. Each actor was associated with a number of movies *n* drawn from the normal distribution with a mean of 5 and standard deviation of 2; once *n* was chosen for a given actor, *n* distinct movies were randomly drawn from the list of all movies generated and associated with the actor. Also normally distributed were the number of words per review (mean = 20, stdev = 5), number of actors per movie (mean = 20, stdev = 8), and number of reviews per movie (mean = 15, stdev = 6). In generating the list of actors associated with a movie, one of the first five actors was chosen randomly 5% of the time, ensuring that each of the first five actors would appear in approximately 1% of all movies; a random existing actor (excluding the first five) was chosen 30% of the time; and a new actor was generated 65% of the time. In no case was the same actor assigned to the same movie multiple times. With 100,000 movies generated, this yielded 1,309,814 unique actors.

JSON records thus generated were written to a text file, with the records subsequently inserted into each of the three database systems. Records were inserted into MongoDB from the generated text file using the *mongoimport* command-line tool packaged with the database system. Records were inserted into PostgreSQL-relational and PostgreSQL-JSON using Python scripts that loaded each record via a separate INSERT query, with all inserts wrapped in a single transaction to increase performance. After inserting data into the systems, appropriate indices were created to increase system performance.

Benchmarking

In benchmarking the insert operations, the insert processes for each of the three database systems were run separately ten times, with all times recorded and used to generate the mean times reported in Table 1. Similarly, for Queries 1 to 6, each query was run ten times, with the mean times reported in Table 1. One set of 100,000 documents was used to benchmark Queries 1 to 5 and the insert operations. Due to

performance concerns, a distinct set of 1000 documents was used to benchmark Query 6. In all cases, Python scripts were used to run queries against the database systems, with elapsed times recorded using Python's *timeit* module.

All tests were conducted on a desktop computer using an Intel Core i5 750 CPU, 8 GB of DDR3 RAM, and 2 TB WD20EARS hard drive. This system ran the 64-bit version of Arch Linux. The following are significant versions of software used:

- Linux kernel 3.17.6
- PostgreSQL 9.4.0
- MongoDB 2.6.6
- Python 3.4.2
- PyMongo 2.7.2
- psycopg2 2.5.4
- ToroDB 0.15
- Java OpenJDK 8.u25

Though all queries were tested against ToroDB, Queries 3 through 6 failed with varying errors, with only the inserts, Query 1, and Query 2 succeeding. Thus, no results are reported for ToroDB.

Query 1

Query 1 undertook a simple select-and-filter operation, finding all movies released in the last year. This task was trivial in each of the three systems. As JSON does not provide a date data type, dates were represented by strings in the “YYYY-MM-DD” format, with all comparisons thus being string comparisons. Though date-specific formats permitting date arithmetic are available in PostgreSQL-relational and MongoDB, their use was avoided to make comparisons across systems as fair as possible.

Query 2

Query 2 found what movies were in both the “action” and “adventure” genres, measuring how efficiently each system could search within the variable-length genre list associated with each movie and select matching documents based on a boolean AND function. Both MongoDB and PostgreSQL-JSON provided array-membership operators allowing this operation to be performed easily. PostgreSQL-relational required joining a given movie to each of its genres, discarding any rows not specific to the “action” or “adventure” genres, then retaining only movies with at least two distinct rows, which corresponded to the two desired genres.

Query 3

Query 3 calculated the mean rating of each movie, stressing the aggregation abilities of each system. In MongoDB, this used the aggregation framework to generate separate documents for each review, then calculate the mean score for each group of documents corresponding to a given movie. PostgreSQL-relational and PostgreSQL-JSON both joined the reviews for each movie to the movie itself, then calculated the mean review score.

Query 4

Query 4 determined which movies contained a given actor, measuring how efficiently the JSON-based systems could search within JSON arrays listing actors for each movie. The actor chosen for each search appeared in 0.1% of all movies, ensuring that the number of result elements would be equal for each system. In MongoDB, this required only a simple array-membership search; in PostgreSQL-JSON and PostgreSQL-relational, the query joined each movie's actors to itself, then selected rows with the desired actor.

Query 5

Query 5 tested the aggregation capabilities of the three database systems by determining which movies had at least half of their reviewers from Canada, requiring quantification of both the total number of reviews per movie and the number of reviews written by Canadian users. Movies without any reviews were ignored. In MongoDB, this required use of a four-stage aggregation pipeline; in PostgreSQL-JSON, this involved joining a given movie to each of the array elements corresponding to its Canadian reviews; in PostgreSQL-relational, this necessitated a three-way join between the *movies* table and two instances of the *reviews* table, with one corresponding to Canadian reviews, and one to all reviews.

Query 6

Query 6 was the most complex query, finding all pairs of movies released at least five years apart that shared at least one actor. Note that the quadratic complexity inherent to this query inhibited its application to the original 100,000-movie dataset—no system produced a result in answer to the query on this dataset within 24 hours. Consequently, these queries were applied to a new dataset consisting of only 1000 movies.

In PostgreSQL-relational and PostgreSQL-JSON, this was done by first generating all pairs of movies released at least five years apart, which required the string-based date column to be converted to a date type to permit date arithmetic. Note that for any two movies *m_1* and *m_2*, duplicate results were avoided by generating only the pair *(m_1, m_2)*, not *(m_2, m_1)*. Subsequently, a correlated subquery was issued for each movie pair to find the intersection of actor sets between them. Only movies whose actor-intersection set possessed at least one member were returned.

In MongoDB, this query was more challenging to perform. First, for a given movie *m_1*, all movies released at least five years earlier were selected. Suppose we designate one such movie as *m_2*. Then, all pairs of actors were examined, such that the first actor came from *m_1*, and the second from *m_2*. In the event these two actors were the same, the number of shared actors in the movie was incremented. Finally, only movies whose shared actor count was at least one were returned. While the Argo authors would have used a map-reduce query in MongoDB for this task (2), this is not possible in MongoDB versions 2.4 and later, whose switch to the V8 JavaScript engine prevents the user from querying collections in the *map* or *reduce* calls.

This query illustrates well the different mindsets underlying the PostgreSQL-based systems and MongoDB. While the PostgreSQL queries retained the declarative nature favoured by SQL, the MongoDB query required a procedural description of how to generate the desired results, making it

considerably more complex. Part of this complexity stemmed from the robust nature of all three queries—they were written to ignore duplicate actors appearing in the actor lists of a given movie, making the system more tolerant of poor-quality data, and to permit selection of movies sharing an arbitrary number of actors, rather than only one.

Results and Discussion

Summation

Performance differed dramatically between systems depending on which query was examined. Both MongoDB and PostgreSQL-relational performed well, with MongoDB emerging as the fastest system for two queries and the insert operations, while PostgreSQL-relational was fastest for four queries. PostgreSQL-JSON performed poorly by comparison—relative to the fastest system in each of the six queries, it exacted performance penalties of 1065x, 25x, 28x, 65x, 14x, and 8x for Q1 to Q6 (Table 1), and was the slowest system for Q1 through Q4. Though both MongoDB and PostgreSQL-relational performed well for the first four queries, PostgreSQL-relational performed poorly in Q5, taking 23x longer than the leading MongoDB, while MongoDB performed poorly for Q6, requiring 11x as long to complete as the leading PostgreSQL-relational.

When all queries were considered, MongoDB performed significantly better than its competitors, taking only 165 s to complete all operations, relative to 463 s for PostgreSQL-relational and 635 s for PostgreSQL-JSON. The dominant factor in this discrepancy, however, was insert time. With insert time excluded, MongoDB took 109.0 s to complete Q1 to Q6, while PostgreSQL-relational took 109.5 s. Despite the almost identical nature of these results, however, performance differed substantially between the systems on individual queries—nowhere was this more apparent than in Q1, where PostgreSQL-relational was 67x faster than MongoDB. While this only amounted to a difference of 0.14 s, it was not so small across all queries. In Q5, for example, MongoDB was 14x faster than PostgreSQL-JSON, causing the former to complete 50 s before the latter.

Notably, Q1 to Q6 can be partitioned into two sets. Q1, Q2, and Q4 all completed in less than 0.1 s in the fastest system, PostgreSQL-relational. Q3, Q5, and Q6 all took at least 3 s to complete in the fastest system, which was MongoDB except for Q6, where PostgreSQL-relational emerged as the best. Thus, we see that relatively simple, fast-completing queries are often best undertaken in PostgreSQL-relational, while more intensive multisecond queries fare better in MongoDB. In particular, the two queries favouring MongoDB made heavy use of aggregation, suggesting that MongoDB may be best for such tasks. Regardless, good performance in PostgreSQL-relational does not predict the same in PostgreSQL-JSON, despite the similarity of the two systems and of the query structures used in each.

Thus, no universal ideal emerges for storing and querying complex JSON documents—one's choice must depend on precisely the nature of document stored, as well as the nature of queries performed.

Query 1

In Query 1, PostgreSQL-relational showed a marked performance advantage relative to the other systems (Fig. 1), answering the query 67.4x faster than MongoDB and 1065x faster than PostgreSQL-JSON (Table 1). Examining the query plans generated by PostgreSQL-relational and PostgreSQL-

JSON suggest why a three-orders-of-magnitude performance difference exists—PostgreSQL-relational performed a bitmap index scan on its movies to find those with release dates in the last year, while PostgreSQL-JSON made no use of its generalized inverted index, electing instead to perform a sequential scan of all movies, then apply the JSON filtering function to each to determine which to return. Though MongoDB fared better than PostgreSQL-JSON, it was still substantially slower than PostgreSQL-relational, suggesting that simple queries drawing on well-chosen indices over relational data will likely be faster than similar queries on JSON data.

Query 2

Query 2 demonstrated similar performance between PostgreSQL-relational and MongoDB (Fig. 2), with the latter only 1.33x slower than the former (Table 1). PostgreSQL-JSON, by contrast, was much slower than both, taking 25x as long as PostgreSQL-relational. Striking is the parallel in performance between PostgreSQL-relational and MongoDB, given that each system utilized substantially different queries—while MongoDB's query used an array membership operator provided by the system, PostgreSQL-relational had to perform both a join and a group operation to answer this query. PostgreSQL-JSON, like MongoDB, used a native array membership operator provided by the database system, yielding a similar query, but with far worse performance than MongoDB or PostgreSQL-relational. This suggests that PostgreSQL-JSON's JSON-related functions are less performant than MongoDB's, perhaps due to their relative immaturity—while MongoDB's function has existed for several years, PostgreSQL-JSON's was introduced only with PostgreSQL 9.4, released in December 2014 (7). Indeed, PostgreSQL-JSON's query plan consisted of only a sequential scan of all movies followed by application of the array membership function, without making use of the generalized inverted index applied to the documents.

Query 3

Query 3 is unique in placing PostgreSQL-relational between MongoDB and PostgreSQL-JSON (Fig. 3)—in all other queries except the insert operations, the two JSON systems were either both faster or both slower than PostgreSQL-relational. In this query, MongoDB furnished a result in 3.26 s (Table 1), with PostgreSQL-relational and PostgreSQL-JSON taking 3.11x and 28.4x as long, respectively. MongoDB's efficient completion likely resulted from use of its well-optimized aggregation framework, while both PostgreSQL-based systems had to perform joins between each movie and its reviews, before grouping the results by movie and computing the average aggregation function on each group. In both PostgreSQL-relational and PostgreSQL-JSON, the most costly part of these operations was the sort necessary for the grouping operation. While the query planner indicates most parts of query execution were comparable in cost between PostgreSQL-JSON and PostgreSQL-relational, PostgreSQL-JSON performed a nested-loop join between the JSON array elements corresponding to reviews, while PostgreSQL-relational used a more efficient merge join. This difference was enough explain why PostgreSQL-JSON took 9 times as long to answer the query relative to PostgreSQL-relational.

MongoDB's advantage relative to PostgreSQL-relational is more difficult to explain, as the aggregation framework in MongoDB performed operations that were conceptually equivalent to those of PostgreSQL-relational—it generated separate movie documents for each review, corresponding to the

join performed in PostgreSQL-relational, before grouping the results by movie and computing the aggregate function. When considered alongside the similarly aggregation-focused Query 5, this suggests that aggregation-centric queries may be significantly faster in MongoDB than either PostgreSQL-relational or PostgreSQL-JSON.

Query 4

Query 4 demonstrated the second-greatest intersystem performance discrepancy of all comparisons (Fig. 4), with PostgreSQL-relational completing 35.1x faster than MongoDB and 64.6x faster than PostgreSQL-JSON. This was unexpected, as both MongoDB and PostgreSQL-JSON provided native functionality for finding documents possessing multiple members of interest in a JSON array, while the relational query had to perform a three-way join between the *movies*, *actors*, and *movies_actors* tables, making the latter appear more complex than the former two.

PostgreSQL-relational's performance advantage may have stemmed from its efficient use of indices and joins. Running EXPLAIN on the PostgreSQL-relational query indicates that it exploited the *actors.id* index to find the desired actor, then was able to perform an efficient hash join between the *movies_actors* and *movies* tables. By contrast, the PostgreSQL-JSON query had to convert the actor IDs in the actors array for each movie to an integer, then compare it to the desired actor ID. Though a single iteration of this operation is cheap with regard to CPU time, performing it for every stored movie is costly, with the query planner indicating the estimated cost of this operation was twenty-fold greater than simply searching the indexed actor ID in the relational query.

Query 5

This query was, like Query 3 and Query 5, among the slowest to execute (Fig. 5). MongoDB demonstrated a significant performance advantage over the two PostgreSQL systems, requiring only 5.1 s to complete the task relative to the 38 s of PostgreSQL-JSON and 48 s of PostgreSQL-relational (Table 1). Conceptually, MongoDB's approach was similar to the PostgreSQL systems'—while a movie with twenty reviews, half of which were Canadian, would generate 1*20*(20/2) = 200 rows over which aggregation occurred in the PostgreSQL systems, the MongoDB query's use of the aggregation framework's *unwind* operator generated a similar multitude of documents.

Query 6

As with the other slow, multisecond queries, Query 6 (Fig. 6) showed substantial performance differences between the different systems (Table 1). Consistent with intuition, MongoDB's complex procedural result generation was an order of magnitude slower than the relational join-and-intersect approach taken in PostgreSQL-relational, with the query times appearing as 9.25 s and 99.9 s, respectively. Surprisingly, however, PostgreSQL-JSON was nearly as slow as MongoDB, taking 73.3 s, despite a query whose structure was similar to that used in PostgreSQL-relational. This performance penalty relative to PostgreSQL-relational likely stemmed from querying fields within the JSON structure, required to retrieve the *title*s and *release_date*s of each movie composing a given pair, or from the *jsonb_array_elements* function used to generate the actor arrays entered in the intersection operations.

Notably, the PostgreSQL-JSON query planner did a poor job estimating total query cost—it indicated a total cost only one-fifth of the equivalent query's cost in PostgreSQL-relational, yet in fact took eight times longer to complete. Of course, it could be that the PostgreSQL-relational query planner was too pessimistic, not that the PostgreSQL-JSON query planner was too optimistic, but PostgreSQL-relational's is likely more accurate, given the system's greater maturity.

Insertion

MongoDB proved fastest in inserting records (Fig. 7), completing 6.38x faster than PostgreSQL-relational and 7.27x faster than PostgreSQL-JSON (Table 1). Several factors may have influenced this advantage. While the *COPY table FROM file* would likely have been faster for the PostgreSQL-based systems, it was not used as it would have necessitated a normalized, specially formatted file for PostgreSQL-relational, meaning that the systems would no longer be using the same JSON file as input and that their results would consequently no longer be directly comparable. Secondly, while the INSERT statements for the PostgreSQL systems were executed by Python code that also loaded and parsed the on-disk JSON records, the records were inserted into MongoDB using the *mongoimport* utility, which is bundled with MongoDB and written in C++ and so likely more performant. Finally, MongoDB is likely better optimized for loading JSON than the PostgreSQL systems, given that JSON is its exclusive focus.

Conclusion

MongoDB and PostgreSQL both offer sufficient expressive power to execute complex queries on JSON documents with intricate structure. They achieve this despite divergent computational models. PostgreSQL-JSON queries are largely similar to queries one would write against a traditional relational schema. By contrast, MongoDB offers an aggregation framework that, for moderately complex queries, is similar to PostgreSQL-JSON; for yet more complex queries, however, MongoDB becomes more difficult to work with, requiring an ad hoc procedural description of how to generate results.

In terms of performance, PostgreSQL-JSON is slow compared to its competitors when executing complex queries, though this may be due to the system's relative immaturity. MongoDB, by comparison, is fast enough to be used in place of relational systems. Nevertheless, performance differs radically from query to query, so end users must benchmark systems against their own workloads to determine which is most suitable. Given time, PostgreSQL-JSON's speed will likely improve, allowing one to store and query JSON alongside relational data using only a single system. Argo and ToroDB are also of interest. If Argo were extended to offer a sufficiently rich query language to express the queries used in this study, its practice of “shredding” JSON documents into relational structures may offer performance benefits relative to systems storing JSON natively, as was seen in the Argo authors' comparison to MongoDB for smaller document sets (2). Likewise, improving ToroDB to support the full range of MongoDB queries would be worthwhile, given that its authors suggest it is significantly faster than MongoDB (8).

This study has demonstrated that issuing complex queries against denormalized JSON documents with intricate structure need not impose a significant performance penalty. Consequently, JSON documents transferred from other systems may be stored directly, without needing to convert them to relational

structures or normalize them after doing so. PostgreSQL-JSON, whose storage, query, and index principles echo those adopted by Oracle Database in integrating JSON support (9), suggests the likely evolutionary path for relational database engines. As JSON evolves from being merely a data transfer format into a means of storing documents against which users wish to execute complex queries, relational database engines will continue to improve support for such tasks, replacing specialized JSON stores such as MongoDB while obviating middleware like Argo and ToroDB.

	Fastest	Second-fastest	Third-fastest	Fastest to 2nd-fastest mag. diff.	2nd-fastest to 3rd-fastest mag. diff.
Insert	MGDB (55.5 s)	PGSR (354 s)	PGSJ (405 s)	6.38x	1.14x
Q1	PGSR (0.00219 s)	MGDB (0.147 s)	PGSJ (2.32 s)	67.4x	15.8x
Q2	PGSR (0.0944 s)	MGDB (0.126 s)	PGSJ (2.40 s)	1.33x	19.1x
Q3	MGDB (3.26 s)	PGSR (10.2 s)	PGSJ (92.8 s)	3.11x	9.14x
Q4	PGSR (0.0687 s)	MGDB (2.41 s)	PGSJ (4.43 s)	35.1x	1.84x
Q5	MGDB (3.93 s)	PGSJ (54.2 s)	PGSR (89.0 s)	13.8x	1.64x
Q6	PGSR (9.25 s)	PGSJ (73.7 s)	MGDB (99.9 s)	7.97x	1.36x
Total	**MGDB (165 s)**	**PGSR (463 s)**	**PGSJ (635 s)**	2.81x	1.37x

Table 1: Elapsed times and performance differences on Queries 1 to 6 and insert operations for fastest, second-fastest, and third-fastest systems on each query. MGDB is MongoDB; PGSR is PostgreSQL-relational; and PGSJ is PostgreSQL-JSON. Recorded times are mean values after ten replicates of each operation.

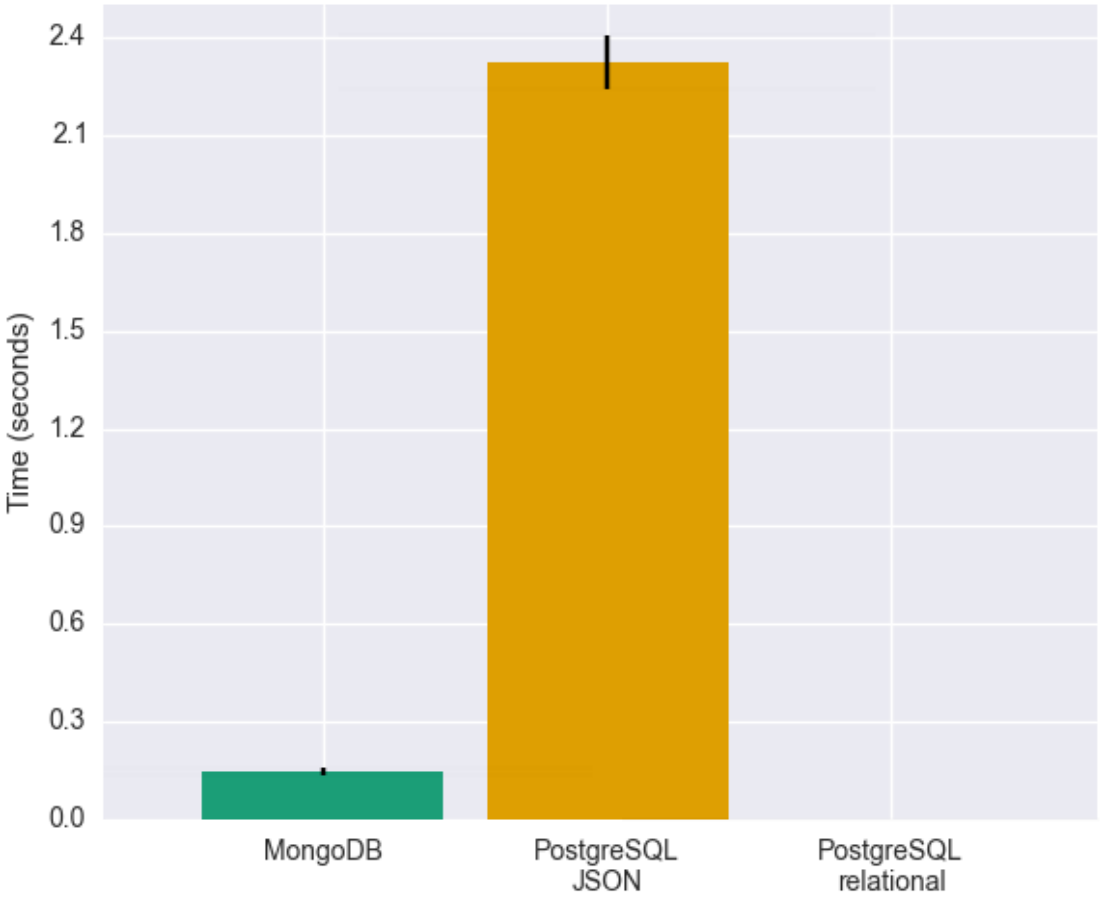

Figure 1: Performance of three systems on Query 1 as mean value recorded after ten replicates of query. Error bars represent standard deviation.

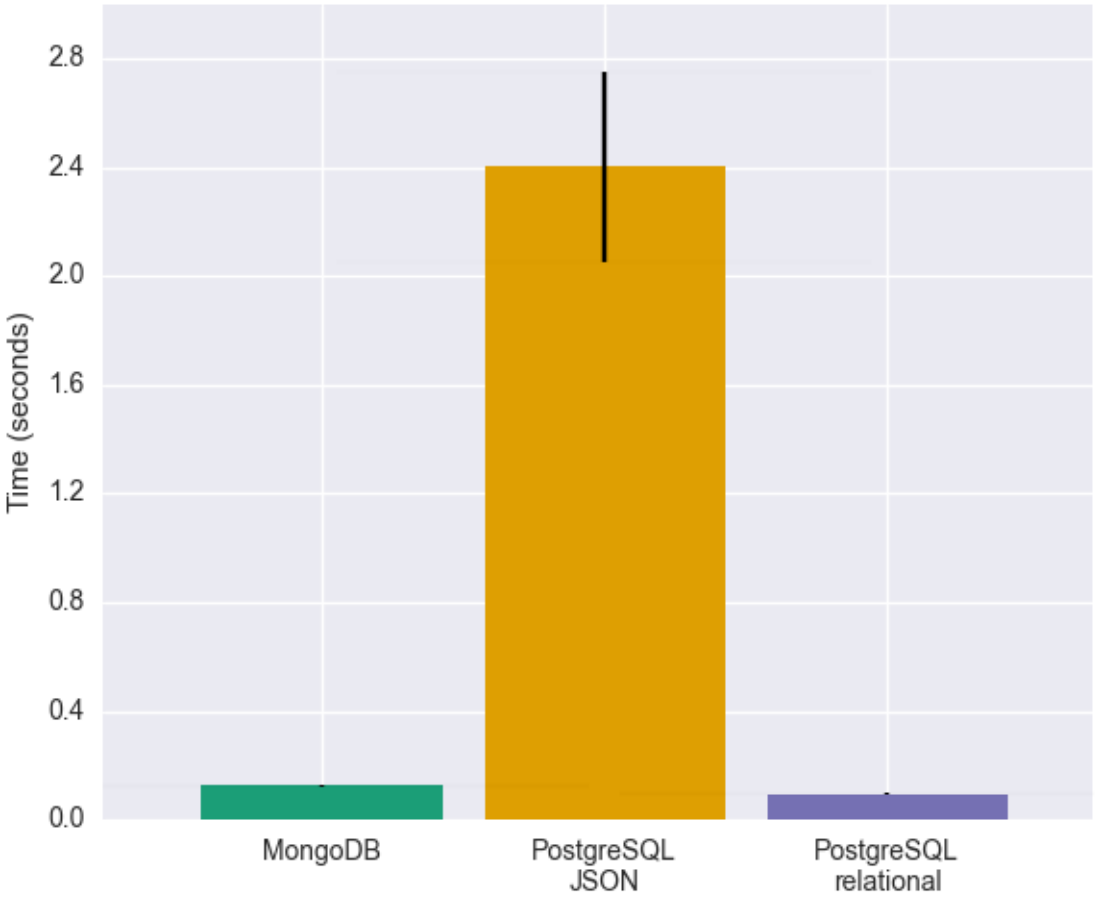

Figure 2: Performance of three systems on Query 2 as mean value recorded after ten replicates of query. Error bars represent standard deviation.

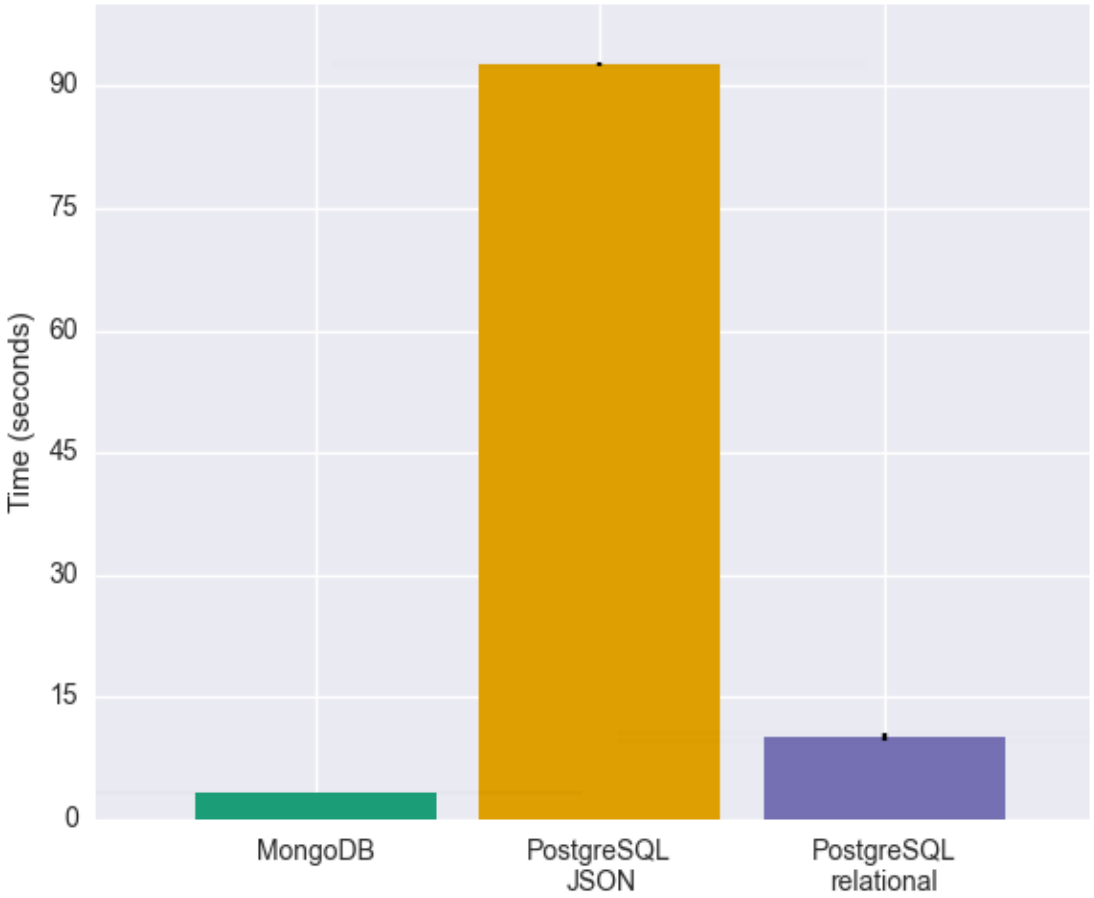

Figure 3: Performance of three systems on Query 3 as mean value recorded after ten replicates of query. Error bars represent standard deviation.

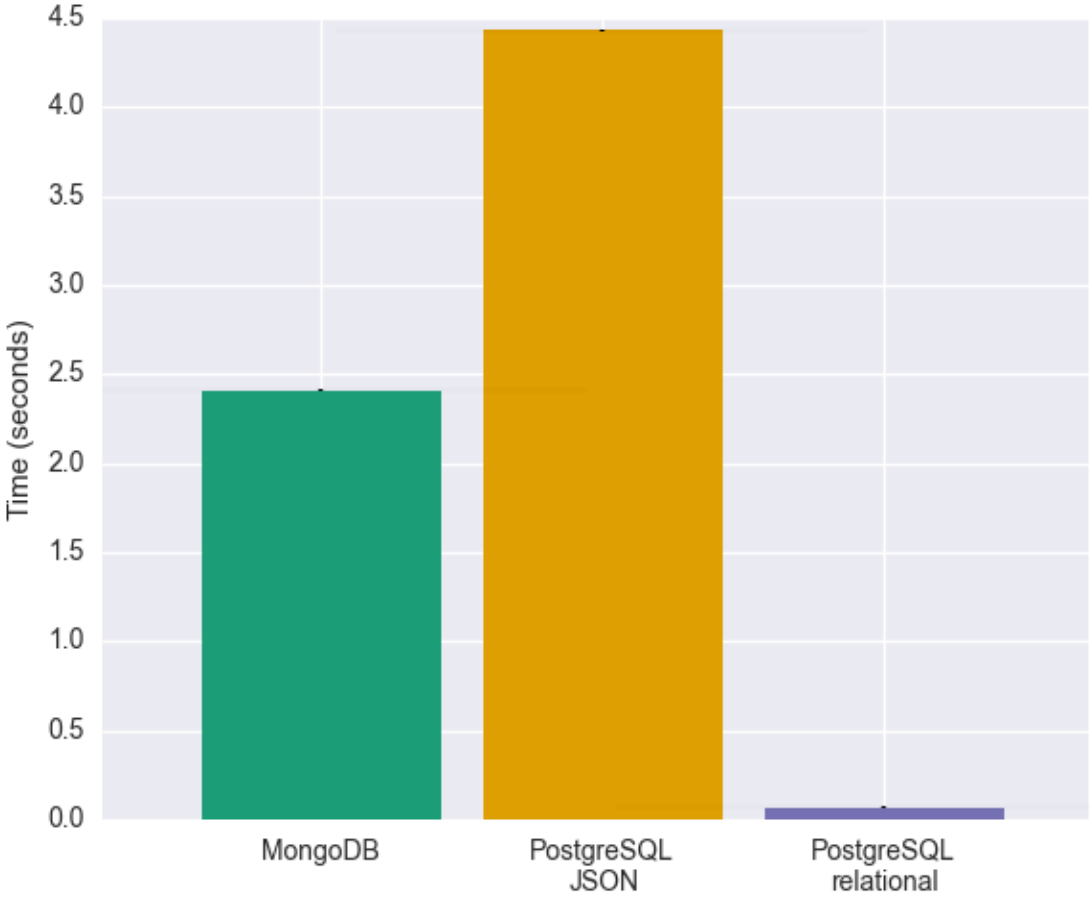

Figure 4: Performance of three systems on Query 4 as mean value recorded after ten replicates of query. Error bars represent standard deviation.

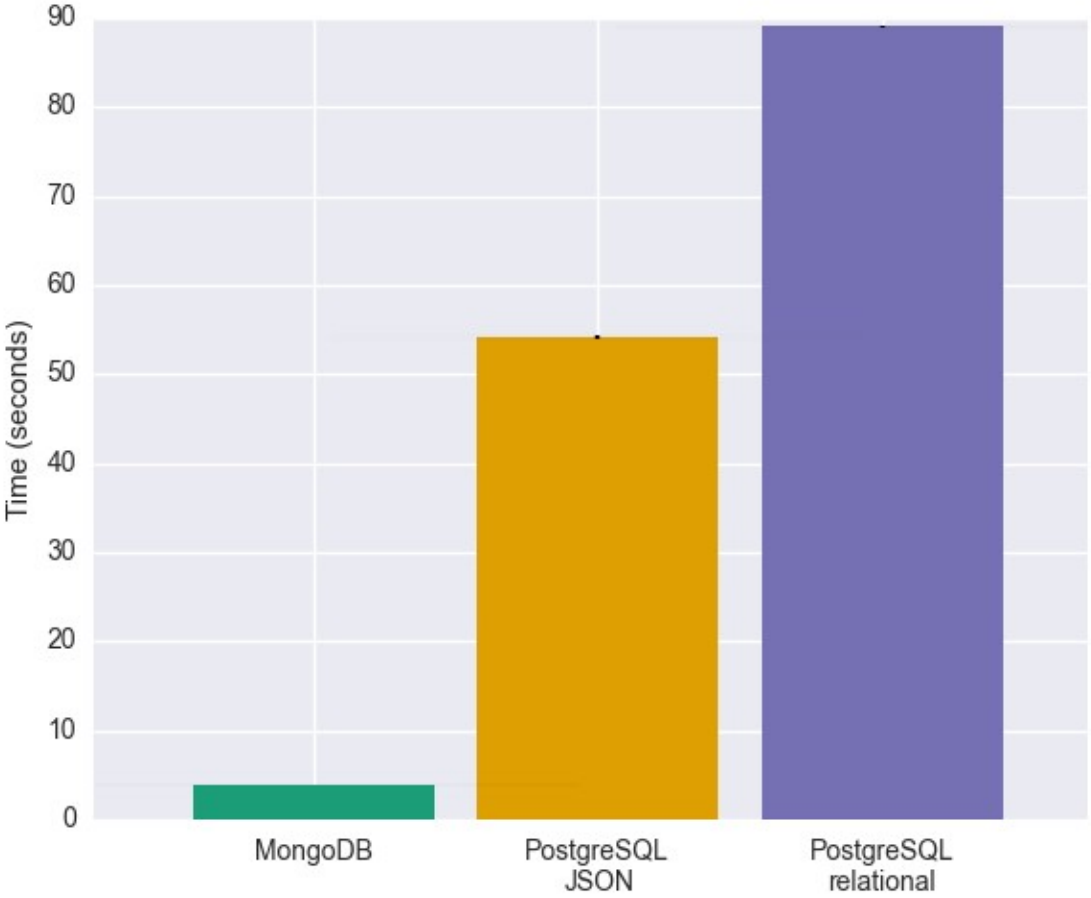

Figure 5: Performance of three systems on Query 5 as mean value recorded after ten replicates of query. Error bars represent standard deviation.

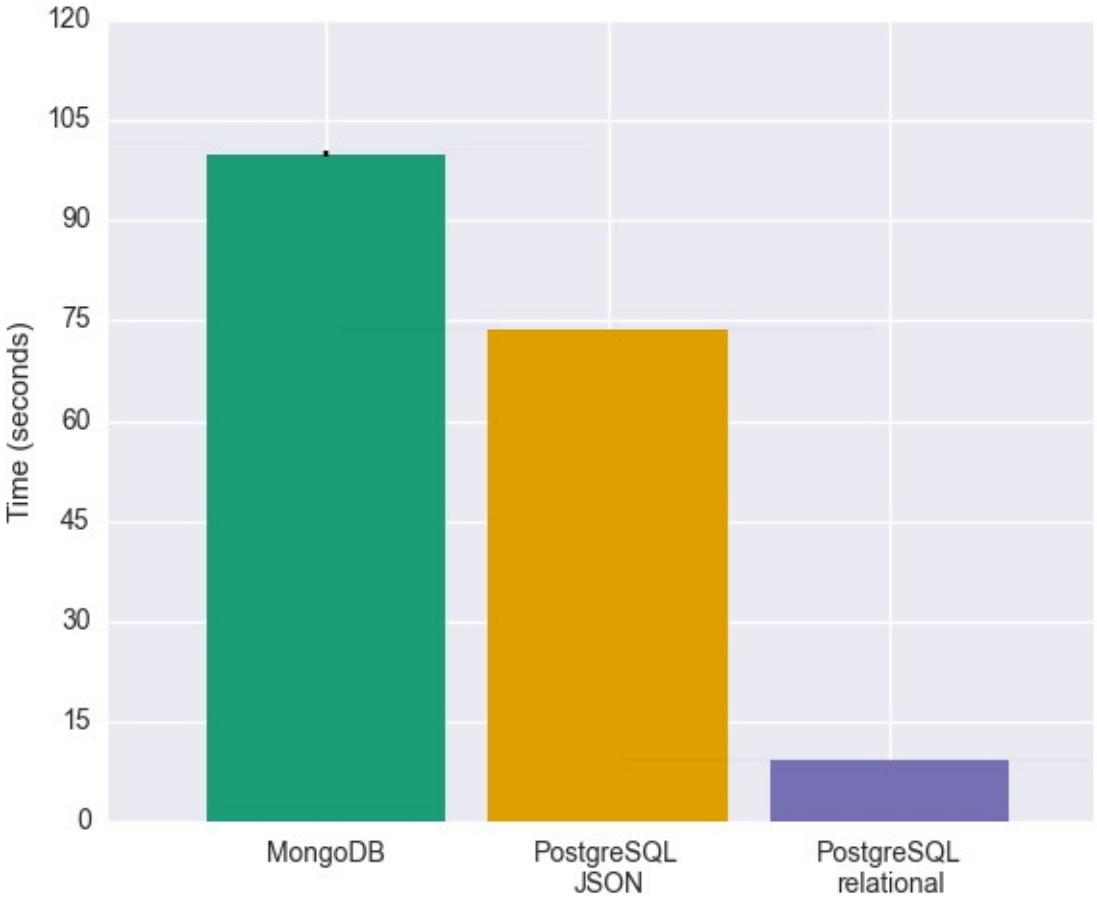

Figure 6: Performance of three systems on Query 6 as mean value recorded after ten replicates of query. Error bars represent standard deviation.

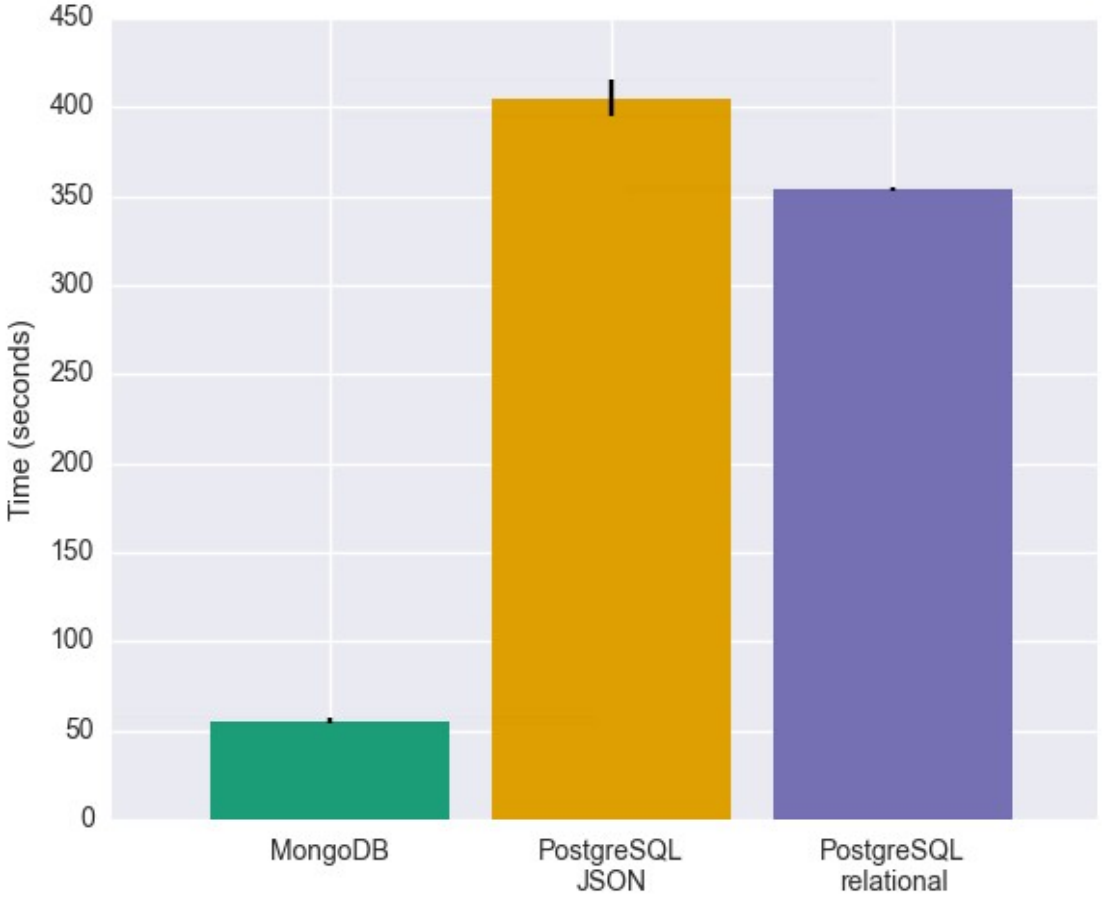

Figure 7: Performance of three systems on insert operations as mean value recorded after ten replicates of inserting same 100,000 documents. Error bars represent standard deviation.

```
{
  "id": 1,
  "genre": [
    "Film-Noir"
  ],
  "title": "XWGPAJ0APRM",
  "actors": [
    {
      "id": 1,
      "birth_date": "1924-03-07",
      "name": "SB31DI4MJYM 9B0FY4WVY4TC5P"
    },
    {
      "id": 2,
      "birth_date": "1903-12-22",
      "name": "TZO71ADT5D NG0D0JBVT0GSON0B9OH"
    },
  ],
  "reviews": [
    {
      "id": 1,
      "rating": 5,
      "user": {
        "recent_purchases": [
          299,
          340,
          405,
          414,
          883
        ],
        "country": "BE",
        "name": "WH1BST9 WDUPB01IEYKYS26"
      },
      "text": "IH5U3 YRHWRGUW9OT6KC11 VR6UBZ22G6G8QDN7 HQUWIB4QPZKPP3TPDFZW
JCE4JKEON60QAFHB6EXS 5OQ5LQ MPQ9H3E YVH N3IWQLGJ6ELNJ JG7IT P5O6J3BQ4M81SKT NP1FXNB
W3CX25GH YVX759VHYIMV6 Q0N8FHRB57TBQD8 R3AHCB9 6B5PT UHPGTVJVQGWU3IXVTK9B
JVURWR1W6PXXG JMLDLOWVL25U4 MOYZ3X1Z4DGXG MDQR2I1G SGF"
    },
  "release_date": "1938-01-09"
}
```


Code Listing 1: Example movie document generated for benchmarks. This document includes randomly generated strings for movie title, actor names, reviewer names, and review text.

References

1. JSON [Internet]. [cited 2015 Jan 5]. Available from: http://json.org/

2. Chasseur C. Enabling JSON Document Stores in Relational Systems. 2013;

3. The Movie Database API [Internet]. [cited 2015 Jan 5]. Available from: http://docs.themoviedb.apiary.io/#reference/movies/movieid/get

4. Rotten Tomatoes API [Internet]. [cited 2015 Jan 5]. Available from: http://developer.rottentomatoes.com/docs/json/v10/Movie_Info

5. Oracle Database 11g Express Edition Release 2 [Internet]. [cited 2015 Jan 5]. Available from: http://www.oracle.com/technetwork/database/database-technologies/express-edition/overview/index.html

6. PostgreSQL 9.3: Documentation: 9.3: Release 9.3 [Internet]. [cited 2015 Jan 5]. Available from: http://www.postgresql.org/docs/9.3/static/release-9-3.html

7. PostgreSQL 9.4: Documentation: 9.4: Release 9.4 [Internet]. [cited 2015 Jan 5]. Available from: http://www.postgresql.org/docs/9.4/static/release-9-4.html

8. ToroDB [Internet]. [cited 2015 Jan 5]. Available from: http://www.8kdata.com/torodb/

9. Liu ZH, Hammerschmidt B, Mcmahon D. JSON Data Management – Supporting Schema-less Development in RDBMS. 2014;1247–58.

10. Rabl T, Gómez-Villamor S, Sadoghi M, Muntés-Mulero V, Jacobsen H-A, Mankovskii S. Solving big data challenges for enterprise application performance management. Proc VLDB Endow. 2012;5(12):1724–35.