CAN RELATIONAL SYSTEMS SUBSUME DOCUMENT STORES? AN EXTENDED BENCHMARK OVER ARGO

Hongyi Wang * **Huawei Wang** * **Yan Nan** *

ABSTRACT

In recent years, Document Store NoSQL gains a huge amount of popularity, which is motivated by supporting JSON data model in a NoSQL system. Although systems like MongoDB and CouchDB already supported JSON document store, a complete support of SQL-like JSON query language seems to lack in those systems. Argo took a first step towards building an automated mapping layer for storing and querying JSON data in a relational system and provided SQL-like JSON query language. Additionally, Argo's benchmark *i.e.* NoBench indicated that Argo outperforms MongoDB on certain tasks. In this project, we put the effort in expanding the empirical analysis of Argo by considering a broader comparison domain *i.e.* comparing with more state-of-the-art Document Store systems over more operations/tasks *e.g.* MongoDB, CouchDB and Elasticsearch. We aim at presenting a more complete benchmark, which might give users potential guidance on trade-off comparisons among systems.

1 INTRODUCTION

NoSQL(Cattell, 2011; Han et al., 2011; Moniruzzaman & Hossain, 2013; Leavitt, 2010; Hecht & Jablonski, 2011) is becoming increasingly popular in recently years due to its no schema convenience and document store nature. Representatives of these systems include MongoDB (mon) and CouchDB(cou). Compared to traditional relational database system, NoSQL are more scalable horizontally and can provide superior performance while can still provide large volumes storage of structured, semi-structured, and unstructured data. Most importantly, these systems are appealing to web developers due to its easiness to generate JSON object without extra processing of underlying data. JSON model is used as a common light weight communication paradigm in many large companys web service. In addition, for those data belonging to other data and always be retrieved at the same time, using JSON model can eliminate lots of useless join accompanied with schema normalization. Finally, JSON model provides more chances for programmers to conduct agile developement. By storing certain data as JSON objects in a column in a relation, a small development team can employ agile techniques to deliver a specific application component in frequent, incremental releases. They can effectively make schema changes but just to their JSON data by modifying the JSON documents themselves directly without changing the relational schema, i.e., execute SQL DDL. Thus, JSON-based document store has a strong appeal to programmers who are working on web services and when speed of deployment is an important issue.

However, NoSQL systems also have some disadvantages compared to traditional relational system. First, the types of queries supported in these NoSQL system is limited. In addition, JSON documents store usually dont keep ACID transaction properties. Furthermore, although the APIs of NoSQL systems are user-friendly, (e.g: in MongoDB, the APIs are very similar to JavaScript), non-negligible gaps still exist. And the learning curve for data analysts without database background remains relatively sharp.

Argo (Chasseur et al., 2013) took a first step towards building an automated mapping layer for storing and querying JSON data in a relational system and provided SQL-like JSON query language. Associated with Argo, a micro-benchmark, called NoBench, was presented to gain insights into various performance aspects of JSON document stores and how Argo compared to them. Based on the empirical results and analysis, the Argo solution is generally both higher performing and more functional (e.g. it provides ACID guarantees and natively supports joins) than MongoDB.

Our Contribution Our contributions in this work are three-folded: i) We reproduced the experiments and extended the original implementation of Argo, which leads to our own variants. In our implementation, all main features of Argo remain the same, but modifications are conducted to improve the performance and flexibility of Argo. Moreover, the backend of Argo is extended. Previously, Argo was built on top of PostgreSQL, we added MySQL backend for Argo to improve it's flexibility.

*Equal contribution . Correspondence to: Hongyi Wang <hongyiwang@cs.wisc.edu>.

Final Project of CS 764 Fall 2018

ii) We walked through number and types of data operations supported in *de facto* database management systems with JSON document store *e.g.* MongoDB, CouchDB, Elasticsearch (Gormley & Tong, 2015) and Amazon DynamoDB (Sivasubramanian, 2012). Empirical studies are conducted to evaluate the performance of these systems on different types of search/queries along with system consistency.
iii) Our empirical study can serve as a user guide to provide high-level picture on trade-offs exist in *de facto* database management systems under different application scenarios.

2 KEYWORD

JSON Document Store, Performance Evaluation, Benchmarks

3 BACKGROUND

JavaScript Object Notation (JSON) model (Crockford, 2006) is being widely used in web services for its convenience and light-weight paradigm. Several relational database systems also already integrate JSON as a possible data format, including PostgreSQL 9.2, Oracle 12c and MySQL 5.7.8. While it's still a bad practice actually to store the JSON format data directly in those relational database system due to the following reasons. First, it makes the query of data very messy because some attributes in the JSON object can be of different types and of different values for the same semantic. Also, a key might not exist in certain rows while appear in other rows' objects. These all make the direct storage of JSON object not a practicable solution.

Following the referenced paper's idea, we will implemented the JSON mapping layer in following method. In order to address sparse data representation in a relational schema as a simple solution, the mapping layer uses a vertical table format, with columns for a unique object id (a 64-bit BIGINT), a key string (TEXT), and a value. Rows are stored in vertical tables only when values are valid for corresponding keys, which enables flexible data object definition without extra overhead. For hierarchical data, we use a natural key flattening approach to represent the keys. Nested object keys are chained together using the special character '.'. For each value in array we use index representation enclosed by bracket to keep consistency in the representation.

Taking the efficiency and simplicity into consideration, we only introduce and reproduce the Argo/3 layer here. The Argo/3 uses 3 separate tables to represent a single JSON collection. Each table has the standard **objid** and **keystr** columns, as well as a **value** column whose type matches the type of data. Specifically, we use TEXT for string, DOUBLE PRECISION for number, and BOOLEAN or BIT for boolean and let those keys in respective tables. This is similar to a previously-studied schema for storing XML (Bray et al., 1997) documents in object-relational databases which uses seperate node tables for different types of XML nodes (element, attribute, and text). Storage in this way can greatly simplify the procedure for query and insertion operation compared to the direct JSON data type storage. When making certain query, rows are read from 3 tables in parallel with the same **objid** to build a single object. If the **objid** has changed, Argo emits the reconstructed JSON object and starts over with a new, empty object. After processing the last row, Argo emits the final object. If there are no rows to reconstruct objects from, Argo simply returns without emitting any objects.

4 IMPLEMENTATION DETAILS

In this section, we provide the details on implementation of our Agro variant, and the approach we took on extending backend of the original Argo implementation (we refer to vanilla-Argo) implementation.

Argo with MySQL backend Our Argo implementation provides an extension of the vanilla-Argo. In the vanilla-Argo, PostgreSQL seems to be the only backend supported. Based on our tests, it fails to connect to MySQL. We took a first step to modify the original code base and add MySQL support for Argo.

Noticed that MySQL differs from PostgreSQL in some perspectives *e.g.* PostgreSQL supports the usage of sequence as a form of object Id of newly added items, in MySQL we can only simulate this behavior using the *AUTO INCREAMENT* columns in another table. We used the fore-mentioned differences as implementation guidance in building MySQL backend for Argo. Our tests indicate it works fairly well. We also successfully deployed MongoDB Community Edition on our experiment machine.

Experimental enviroment All experiments in this project are planed to be conducted on CloudLab(Ricci et al., 2014) server. Specifically, the machine is a single physical node of c220-g2 and has a main memory capacity of $160GB$.

Challenges and solutions in reproducing *NoBench* The first challenges we met is the disk space limitation of virtual instances provided by CloudLab. The issue was resolved by reassigning the disk space mounted on fast SSD, expanding it up to 440GB such that it can hold up to 64 million json objects documents declared in the original paper.

Another issue we met was to load the Argo splitted tables into PostgreSQL/MySQL. The original paper only declares to use the original tools provided by RDBMS such as the

COPY table FROM file (1)

and

LOAD DATA LOCAL INFILE file
REPLACE INTO TABLE table; (2)

However, it seems that these tools are designed for loading structured data into database and it is difficult to reformat the original JSON collections into respective file. One naive solution is to use a loop and execute the originally provided $INSERT$ SQL sentences multiple times. However, this approach will certainly incur database multiple database connection trials and cause heavy overhead for the network I/O. Our solution is to cache some batch size of documents field in memory and then make use of the SQL sentence of

INSERT INTO database VALUES (val1, val2, val3),
(val1, val2, val3)

which is enabled by most of the RDBMSs in data loading for the purpose of reducing network connection overhead. Although it is still slow compared to the MongoDB batch import tools, we finally successfully imported a scale factor of 16 million JSON objects.

5 EXPERIMENTS

We deployed these following experiments in order to evaluate the performance of these systems under different workload and provide a relative comprehensive user guidance.

For each query, 7 runs are performed. Each individual run of the queries Q3-Q9 indicated in Figure 5 has its parameters randomized so that the results are not distorted by caches in the database system. The results reported bellow are "warm" queries. We discard the maximum and minimum values among the 10 runs for each query, and report the average runtime of the last 5 runs.

Implementation and Setups We follow the original Argo paper to implement the Argo layer on top of PostgreSQL and extend the their implementation on top of MySQL. Our experimental pipeline is deployed on a CouldLab server equipped with two Intel E5-2660 v3 10-core CPUs at 2.60 GHz, 160 GB DDR4 2133 MHz dual rank RDIMMs, and one Intel DC S3500 480 GB 6G SATA SSDs. We use the data generator provided in NoBench directly to generate JSON objects with the same attributes.

Queries In the preliminary experiments, we reproduce the 10 queries/operations designed in NoBench. These operations includes normal projection on one fields and multiple fields, query with single predication, query with range search and bulk writing. We generate JSON documents using nobench with the scale factor of 1 million, 4 million and 16 million JSON documents respectivly. While the 64 million objects documents failed to load into MongoDB due to the main disk partition is not big enough. Specifically, we have finished conducting following experiments.

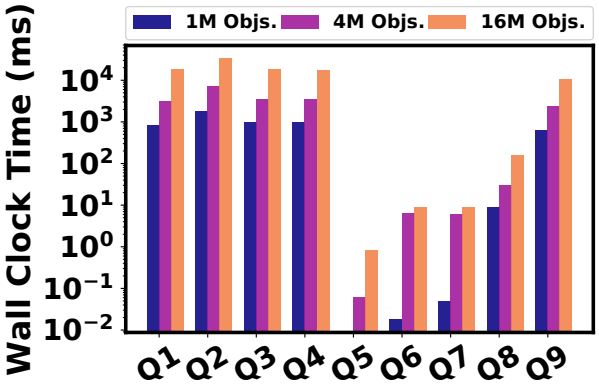

Figure 1. Runtime of Nobench query set for MongoDB

Q_1: projects two common attributes from all the objects in the database, Q_2: projects two nested attributes from the entire dataset, Q_3: projects two sparse attributes from one of the 100 clusters, Q_4: projects two sparse attributes from two different clusters, Q_5: single object selection using matched predicate, Q_6: selects 0.1% of the objects in the collection via a numeric range predicate, Q_7: selects 0.1% of the objects in the collection via a numeric range predicate for a particular range, Q_8: selects approximately 0.1% of the objects in the collection by matching a string in the embedded array, Q_9: same as Q_8 but on a sparse attribute.

1. The running time of NoBench on MongoDB community edition. We choose the scale factor of 1 million objects, 4 million objects and 16 million objects correspondingly. In order to balance the parameters' setting and follow the guidance of MongoDB manual user page, we construct index on frequently used fields including str1, num and nested_arr. The running time figures are shown below. According to the run time statistics, it can be seen that constructing index on specific fields can also greatly speed up the execution of queries on MongoDB document collecitons. However, one limitation about MongoDB is that when building indexes on certain collections, By default, creating an index on a populated collection blocks all other operations on a database. When building an index on a populated collection, the database that holds the collection is unavailable for read or write operations until the index build completes. Any operation that requires a read or write lock on all databases (e.g.listDatabases) will wait for the foreground index build to complete. This may limit the construction of index to only maintenance periods in an actual production environment. Besides, it is naturally to expect that the the average index construction time will enlarge significantly with

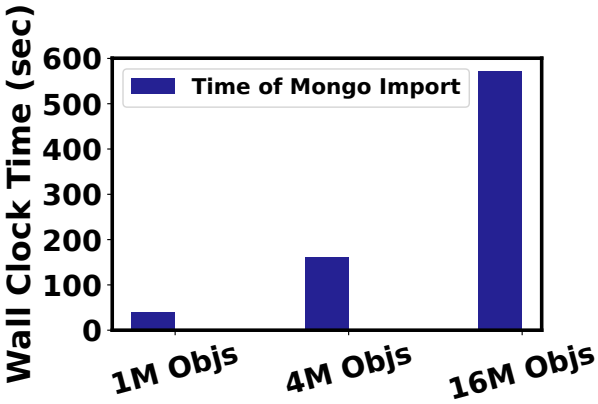

Figure 2. Runtime of Bulk writing for MongoDB

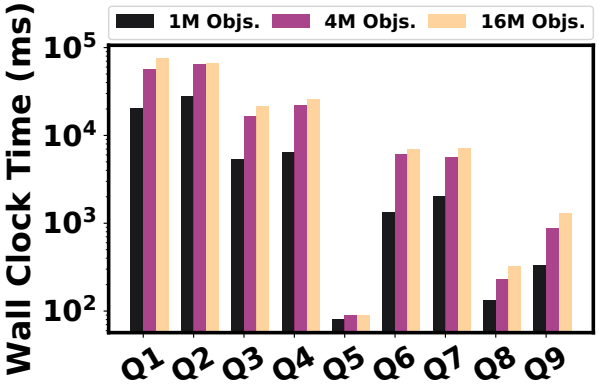

Figure 3. Runtime of Different Query on Argo/MySQL

the increasing of scale factors. Finally, the MongoDB even fails to build too many indexes on the collections as due to some scalability problem in our experiments. Another shortcoming in MongoDB we currently found is that it doesn't support concurrent query accompanied with the bulk writing, which can be achieved by adjusting the isolation level in RDBMS. We may develop some other query set in order to test its performance.

2. The running time of NoBench on Argo/Postgresql and Argo/Mysql. The experiment results proved that the performance is worse but very similar to MongoDB. The gap enlarges especially for the bulk writing operations. We analyze the reason and owe it to the differences of the storage format in these systems. Because the storage format on MongoDB is a continuous storage which eliminates the requirement for decomposition and make use of space locality. On the contrary, the write in Argo need to conduct operations on 3 tables respectively, which will cause operation overhead and also can't make use of locality property during query operation.

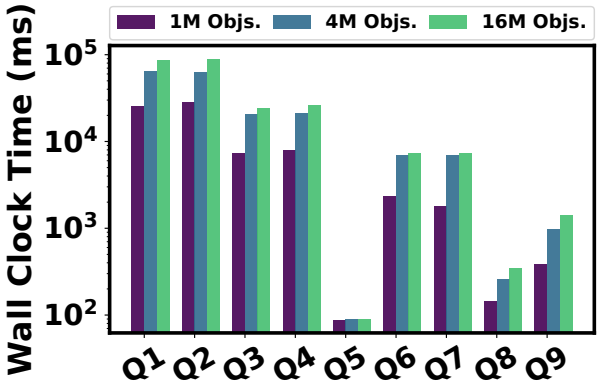

Figure 4. Runtime of Different Query on Argo/Postgres

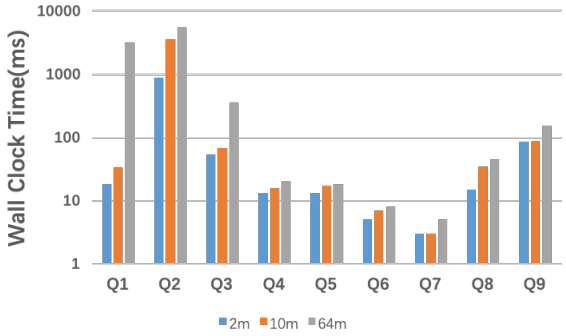

Figure 5. Runtime of Different Query on ElasticSearch

3. The running time of partial NoBench query on ElasticSearch. Because ElasticSearch is designed specifically as a distributed search engine, we expect it to have higher query performance in searching documents. The experiments result show that for the same query and scale of data objects, the speed up factors of ElasticSearch is 10-30 times to MongoDB. Furthermore, it also provides complex data analytics query capability. However, this engine doesn't provide RDBMS's transaction semantics and it not very suitable for consistent data storage. The server documents' size become so large when the number of document grows because the huge number of inverted indexes constructed. The search performance also degrade with the increasing of JSON documents, the best practice is usually to clear some unused documents and also shards the documents into different nodes.

6 Conclusions and Future Work

6.1 Conclusion User guide

float

We have provided a relatively comprehensive benchmark

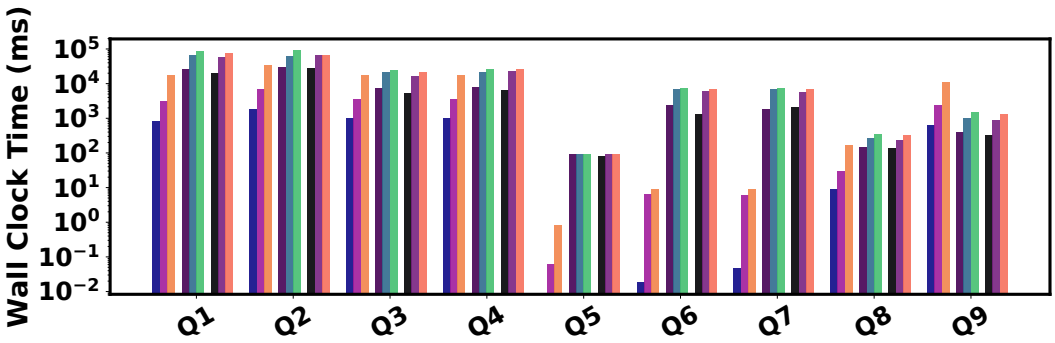

Figure 6. Comparison for 3 different systems

Performance	bulk write	query	functionality	Application
Argo/MySQL	Slow (Needs improved in future)	acceptable, worsen when data scales up	RDBMS capabilities; ACID; just SQL	Good practice for RDBMS. Keep ACID semantics
MongoDB	Specific tools provided for JSON array loading	good, worsen when data scales up	support complex query; JavaScript form API	Lightweight web service, persistent storage for documents.
ElasticSearch	Acceptable	Very fast	Complex aggregation; RESTful API && Java API	Search engine for data analytics

Figure 7. Summary

on 4 different systems, including MongoDB, Argo/Postgres, Argo/MySQL, ElasticSearch. They both have the functionality of storing JSON document and making complex queries. Our experiments show that ElasticSearch can provide the best speed in making certain query, which is accompanied with its generated large index and document size. Argo can provide almost same query speed with MongoDB, also their generated database size doesn't differ from each other very much.

In summary, the implemented Argo transformation layer is a good practice in that it enables the JSON document store in relational database system. User can add more functionality to make fully utilize of the ACID semantics of RDBMS, which is not owned by NOSQL system. NOSQL product is very suitable for web service development due to its easiness to transform objects from web api. However, it may not be appropriate for those applications requiring strong consistency and workload-heavy transactions. Finally, some new-emerging "database system" is worthy considering under certain application scenarios, such as Casandra(cas), ElasticSearch and Redis. Here we just pick up ElasticSearch and show its advantage in querying documents with great speed. Finally, we list out a summary table to for comparison and also a reference for users to choose between these 4 systems.

6.2 Future Work

- Currently, loading large number of JSON documents into the database using Argo is slow. We have tried to use a single transaction to organize all single insert SQL. But it is still too slow. In the future, We may try to make full utilize of the provided bulk writing ability of the underlying RDBMS to perform higher efficiency data loading. In order to use this method, we may have to reorganize the JSON data into the corresponding structured format. The data scale we can achieve using these methods is also still unknown and we want to lift it as much as possible in order to make a fair comparison with NoSQL products.

- Much more NoSQL products' performance can be tested to expand this benchmark.

REFERENCES

Manage massive amounts of data, fast, without losing sleep. http://cassandra.apache.org/. 2018.

Apache software foundation. apache couchdb. http://couchdb.apache.org/. 2011.

10gen, inc. mongodb. http://www.mongodb.org. 2011.

Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. Extensible markup language (xml). *World Wide Web Journal*, 2(4):27–66, 1997.

Cattell, R. Scalable sql and nosql data stores. *Acm Sigmod Record*, 39(4):12–27, 2011.

Chasseur, C., Li, Y., and Patel, J. M. Enabling json document stores in relational systems. In *WebDB*, volume 13, pp. 14–15, 2013.

Crockford, D. The application/json media type for javascript object notation (json). Technical report, 2006.

Gormley, C. and Tong, Z. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine*. " O'Reilly Media, Inc.", 2015.

Han, J., Haihong, E., Le, G., and Du, J. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pp. 363–366. IEEE, 2011.

Hecht, R. and Jablonski, S. Nosql evaluation: A use case oriented survey. In *Cloud and Service Computing (CSC), 2011 International Conference on*, pp. 336–341. IEEE, 2011.

Leavitt, N. Will nosql databases live up to their promise? *Computer*, 43(2), 2010.

Moniruzzaman, A. and Hossain, S. A. Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *arXiv preprint arXiv:1307.0191*, 2013.

Ricci, R., Eide, E., and Team, C. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

Sivasubramanian, S. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pp. 729–730. ACM, 2012.