Evaluating Argo on Data with Increased Complexity

Christopher A. Cremer
1001140650

***Abstract*— Argo is an automated mapping layer that runs on top of a traditional RDBMS, and presents the JSON data model directly to the application/user. Argo has been previously shown to have a significant speedup over document-oriented databases, specifically MongoDB. This was shown by evaluating the systems using the benchmark NoBench. This benchmark evaluates queries on relatively simple JSON documents. Therefore, in order to gain a more comprehensive assessment of Argo, I extended the NoBench data to include more complex data, such as highly nested objects and arrays with more elements. Here I show that increasing the complexity of the data reduces the speedup that Argo has over MongoDB.**

***Index Terms* — Argo, JSON, RDBMS, PostgreSQL, MongoDB**

I. INTRODUCTION

JSON is a dominant standard for data exchange among web services therefore giving a JSON-based document store a strong appeal to programmers who want to communicate with popular web services in their applications [1]. Document store NoSQL systems such as MongoDB and CouchDB support JSON (Javascript Object Notation) as their data model. This data model fits naturally into the type systems of programming languages like Javascript, Python, Perl, PHP, and Ruby. NoSQL document stores use the flexibility of JSON to allow the users to work with data without having to define a schema upfront. This flexibility eliminates much of the hassle of schema design, enables easy evolution of data formats, and facilitates quick integration of data from different sources.

Even though JSON document stores have many advantageous properties, they have some drawbacks when compared to traditional relational DBMSs. One drawback is that the querying capability of these NoSQL systems is limited – they are often difficult to program for complex data processing, and there is no standardized query language for JSON. In addition, leading JSON document stores don't offer ACID transaction semantics. NoSQL systems typically offer BASE (basically available, soft state, eventually consistent) transaction semantics. The BASE model aims to allow a high degree of concurrency, but it is often difficult to program against a BASE model.

Due to these drawbacks, Chasseur et al. [1] designed and evaluated an automated mapping layer called Argo. Argo runs on top of a traditional RDBMS, and presents the JSON data model directly to the application/user, with the addition of a sophisticated, easy to use SQL-based query language for JSON. Therefore, with Argo we gain a highly usable query language and additional features (including joins, ACID transactions) that are naturally enabled by using an underlying relational system. The Argo storage format was designed to be as simple as possible while still being a comprehensive solution for storage of JSON data. One major technical challenge comes from varied structures of records caused by the schema-less data model and schema evolution [2]. They evaluated two storage schemes which they call Argo/1 and Argo/3 inspired by previous work which introduced a vertical representation for sparse data in relational systems [3]. Argo/1 uses a single-table vertical format and retains the type information inside the table. Argo/3 uses three separate tables (one for each primitive type) to store a single JSON collection. There are indefinitely many possible storage formats, and they do not claim that their two schemas are optimal since experience with XML suggests that the best-performing format is application-dependent. Their results demonstrated that the Argo solution is generally both higher performing and more functional than just a document store. For instance, all queries that they ran showed a speed up when using Argo versus a document store database such as MongoDB.

Thus the point of this project is to verify the results found in the Argo paper. Furthermore, since one of the main

challenges is converting complex data to fit into a relational table, I expand their tests by making more nested data as well as longer arrays and comparing the speed of Argo versus a document store. Just like in their paper, in this project I implement Argo on PostgreSQL and compare it to MongoDB. To evaluate Argo, I use NoBench which is a benchmark suite that evaluates the performance of several classes of queries over JSON data in NoSQL and SQL databases. This project begins with replicating their experiments then testing the systems with more complex data.

II. Materials And Methods

A. Experimental Setup

I ran the experiments on a server with 64 Intel Xeon E7-4830 cores at 2.13GHz with 2MB of RAM. The system runs 64-bit Red Hat 4.4.7 (kernel 2.6.32). MongoDB version 3.0.4 was used and PostgreSQL (version 9.4.4) was used for Argo. For each query, 10 runs are performed. The maximum and minimum values among the 10 runs for each query are discarded and the mean execution time of the remaining 8 runs is reported.

B. NoBench

This benchmark aims to identify a simple and small set of queries that touch on common operations. It also generates data which includes hierarchical data, dynamic typing, and sparse attributes.

C. More Complex Data

To increase the level of nesting of the data, I made a new data set similar to NoBench but I also added an object to each document that is 7 levels deep. At each level of nesting of this object, there is a string and an integer. This is referred to as nested NoBench dataset.

I also made another data set that contained longer arrays. In NoBench, the longest any array can get is 7. For the new dataset I added arrays that were between 1 and 30 objects long. This dataset is called long-array NoBench dataset. Essentially, the data is the same as regular NoBench, but for each complex dataset I've added an object that is either more nested or an array that is long.

D. Number of Documents

I chose to run my experiments on 1 million JSON documents rather than 4, 16, or 64 million as done in the paper because with 1 million we see the largest speedup (see Fig 3 of [1]). For the same reason, PostgreSQL is used over MySQL, since the speedup is greater in PostgreSQL compared to MySQL. An additional reason to use less data is that the time and space requirements are reduced.

For the experiments that use more complex data, I only used 100 thousand documents. This was done because the increased complexity also increased the amount of data required to store the dataset. This is amplified by the schema of Argo since there is a significant amount of replication of data. For example, inserting 1 million documents of the complex data took over 16 hours to insert into PostgreSQL.

E. Iteration Over Query Outputs

As can be seen in my code, I iterate over the outputs of each output of the queries. This is done because Mongo returns its output lazily, meaning it returns a cursor for the first 20 results, which ends up being very fast. Thus to obtain an accurate estimate of the run time, I iterate over all the output so that it must actually return a cursor to each row. Since this is done with MongoDB, I also do the same procedure with the PostgreSQL outputs.

III. Results

A. NoBench Results

First, I looked at replicating the results found in [1]. In the paper they found that the schema of Argo/3 was superior to that of Argo/1. Comparing the purple and red bars in Fig 1, we see that Argo/3 consistently has a lower running time than Argo/1. Second, the authors found that Argo/3 was superior to MongoDB for all queries. Again, in Fig 1 we see that Argo/3 takes less time on all queries compared to MongoDB. However, for queries 1 and 2, the magnitudes of the speedups in the paper are not consistent with mine. For instance, the paper found that for query 2, Argo/3 had a speedup of nearly 50x compared to MongoDB. Whereas I found that the speeds were very comparable; the speedup was 1.4x, as seen in Table 1.

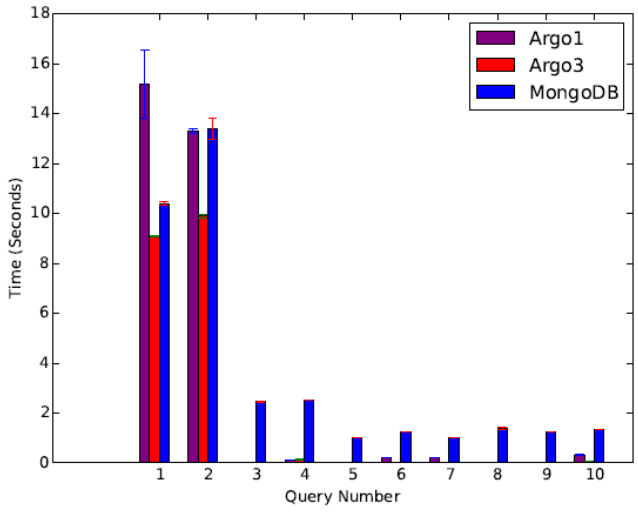

Fig 1: Running times of queries 1-10 on 1 million NoBench documents. The maximum and minimum values among the 10 runs for each query are discarded and the mean execution time of the remaining 8 runs is reported.

Thus my results are consistent with that of the original authors but they do not agree on the magnitudes of difference. Given the overall better performance with Argo/3 compared to Argo/1, for the remainder of this paper I only consider the Argo/3 mapping.

B. *Nested Results*

Next I looked at how the speedup would change when the data consisted of more complex data. In this case, the increased complexity comes from data that has more nesting than the original NoBench data. The result is that for every query there was a decrease in the speedup compared to regular NoBench data. For regular NoBench data, the geometric mean and arithmetic mean of the speedups between Argo/3 and MongoDB were 49x and 314x, respectfully. In contrast to 29x and 135x for NoBench data with nested objects (see Table 2).

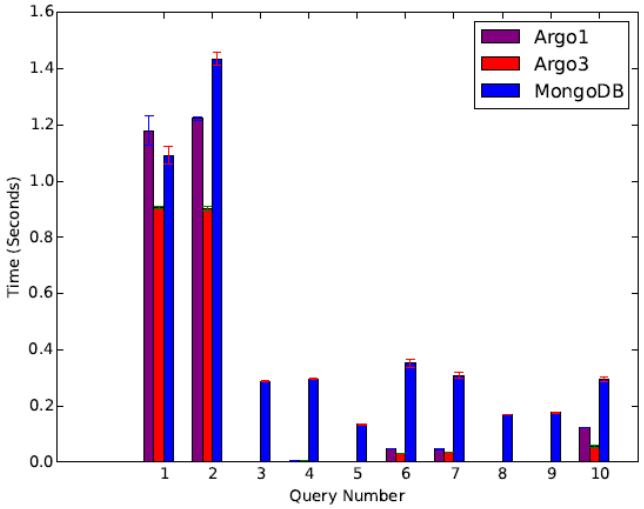

Fig 2: Running times of queries 1-10 on 100 thousand NoBench documents with an additional nested object. The maximum and minimum values among the 10 runs for each query are discarded and the mean execution time of the remaining 8 runs is reported.

C. *Array Results*

Similar to the nested experiments, I looked at increasing complexity but instead of increasing the nesting of the data, I added an array with more elements than NoBench had previously. Interestingly, the drop in speedup was even more significant than the nested data. The geometric mean went from 49x to 25x and the arithmetic mean went from 314x to 90x on regular data compared to data with increased array lengths (see Table 3).

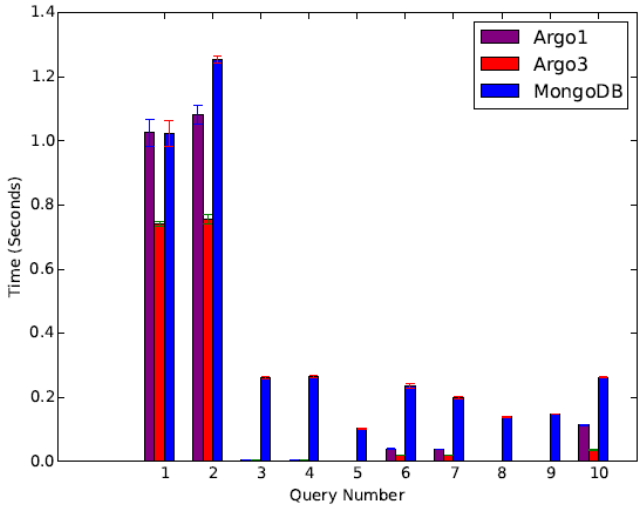

Fig 3: Running times of queries 1-10 on 100 thousand NoBench documents with an additional array object. The maximum and minimum values among the 10 runs for each query are discarded and the mean execution time of the remaining 8 runs is reported.

IV. Discussion

A. *Increasing Complexity Reduces Speedup*

As seen in the results section, increasing the complexity of the data reduced the speedup that Argo/3 has over MongoDB. This is significant because it shows that Argo/3 is less suited for complex data and its advantages are highly data and query dependent. The main reason for the change in speedup is that the nested values are handled with key-flattening and that MongoDB and Argo differ in their storage format. MongoDB stores whole JSON objects contiguously using a fast binary representation of JSON. Argo decomposes JSON objects into a 1-row-per-value format. Also, indices in MongoDB are on the values of a particular attribute. Indices in Argo are built separately on objid, keystr, and various column values. MongoDB indices therefore tend to be smaller and are specialized to particular attributes

B. *Queries 1 – 4*

The queries 1 – 4 are projection queries. The speedups of queries 1 and 2 remained relatively constant for all types of datasets since these projections were performed on every object. Interestingly, queries 3 and 4 actually showed an increase speed up in the more complex data. Query 3 tests the performance when projecting attributes that are defined in only a small subset of objects. Query 4 retrieves the same number of attribute values as Q3, but fetches them from twice as many objects (i.e. the cardinality of the result of Q4 is twice that of Q3). This reduction in speedup is unexpected because Argo can use an index to quickly fetch just the appropriate rows, while MongoDB must scan through every object and project two attributes out from each. The change in

speedup must be related to the difference in number of rows that the Argo mapping creates.

C. Queries 5 – 9

The queries 5 – 9 are selection queries. Some select individual rows and others select about 0.1% of the data. They all showed a decrease in speedup when using complex data instead of the regular data. This is likely caused by the difference in indices of the two systems. MongoDB has indices on the values of a particular attribute whereas Argo has indices on objid, keystr, and various column values. Therefore increasing the complexity of the data significantly affects selection queries.

D. Queries 10

Query 10 is an aggregation query. This query selects 10% of the objects in the collection and COUNTs them. This query tests the performance of the COUNT aggregate function with a group-by condition. The speedup went from 29x in normal data to 5-7x in complex data. Therefore the increase in complex data significantly affected the aggregation speed.

E. Query 11 and 12

Query 12 tested the performance of inserting data, and measures the cost of object decomposition and index maintenance. This query inserts objects, in bulk, amounting to 0.1% of total data size. Since they only measure the insert time of 0.1% of the data, this query is misleading. In the Argo paper, they report the insert time for MongoDB to be 0.19 seconds and 0.83 seconds for PostgreSQL resulting in a speedup (or speed down) of 0.23. However, the difference in my experiments was much more significant. MongoDB inserts were on the order of seconds whereas PostgreSQL was hours. This is a significant difference that should be taken into account. However, I did not include this query in my analysis because I was interested in the changes of projections, selections, and aggregations with the more complex data. For query 11, it required that a JavaScript MapReduce function be implemented but the authors did not provide the necessary code. Therefore it was avoided and is an area for future work.

F. Arithmetic Mean vs Geometric Mean

In the Argo paper, the comparison of the different systems is quantified by the geometric mean of the various queries. The geometric mean of speedup factors does not, therefore, indicate that the combination of Argo and a particular database is definitively faster or slower than MongoDB. I chose to report both the arithmetic and geometric mean because it gives a better representation of the changes of speed ups. Also, the arithmetic mean seems to show larger changes than the geometric mean. Also, NoBench queries are dominated by selection (five queries) and projection (four queries), thus the reported means will be more heavily influenced by these classes of queries than joins, aggregates, and inserts. The mean of speedup factors is useful for making broad at-a-glance comparisons between systems, but, as we saw, fully understanding the differences in performance between document stores requires looking at the results for the individual queries

V. Conclusion and Future Work

My evaluation shows that Argo/3 generally offers performance superior to Argo/1. In addition, Argo/3 is generally superior to MongoDB given regular NoBench data. However, I showed that with increased nesting or longer arrays, the speedup decreases. I also showed that depending on the type of selection query, MongoDB can be superior to Argo. As the original Argo paper mentioned, the optimal schema is highly data and query dependent, therefore we need to look into defining a new schema that is optimal for highly complex data. Even if the Argo schema does not provide any speedups, the fact that it is implemented on a relational database such as PostgreSQL, means it provides SQL queries and supports joins. Another benefit is that it provides ACID guarantees which in some circumstances may be a necessary characteristic that is not possessed by document-oriented databases. One possible downside to keep in mind is that there is a significant insert cost when using the relational database compared to a document-store. Other areas of research should be aimed at expanding the current evaluation to explore the impact of multi-core and cluster environments, as well as evaluating distributed key-value stores and alternative relational mapping schemes. As a whole, the performance of the Argo schemas make them a compelling alternative to document-oriented databases for storing JSON data.

References

[1] C. Chasseur, Y. Li, and J. Patel, “Enabling JSON Document Stores in Relational Systems,” *Sixt. Int. Work. Web Databases (WebDB 2013)*, 2013.

[2] L. Wang, O. Hassanzadeh, S. Zhang, and J. Shi, “Schema Management for Document Stores,” vol. 8, no. 9, pp. 922–933, 2015.

[3] R. Agrawal, A. Somani, and Y. Xu, “Storage and querying of e-commerce data,” *VLDB*, pp. 149–158, 2001.

VI. SUPPLEMENTAL

Query	Argo/1	Argo/3	Mongo	**SU**
1	14.623	9.818	10.583	**1.078**
2	14.530	9.814	13.663	**1.392**
3	0.036	0.035	2.340	**65.974**
4	0.124	0.116	2.419	**20.840**
5	0.001	0.001	0.900	**1643.050**
6	0.242	0.025	1.195	**47.331**
7	0.241	0.023	0.983	**42.744**
8	0.002	0.002	1.378	**779.873**
9	0.003	0.003	1.358	**515.907**
10	0.348	0.045	1.323	**29.433**
GM				**49.087**
AM				**314.762**

Table 1: Runtimes of 1 million regular NoBench objects

Query	Argo/1	Argo/3	Mongo	**SU**
1	1.180	0.906	1.092	**1.206**
2	1.223	0.902	1.436	**1.592**
3	0.004	0.003	0.288	**92.438**
4	0.004	0.004	0.296	**74.267**
5	0.000	0.000	0.134	**453.661**
6	0.048	0.030	0.353	**11.841**
7	0.047	0.034	0.309	**9.205**
8	0.001	0.001	0.170	**286.713**
9	0.001	0.000	0.176	**411.982**
10	0.123	0.055	0.294	**5.328**
GM				**28.927**
AM				**134.823**

Table 2: Runtimes of 100 thousand nested NoBench objects

Query	Argo/1	Argo/3	Mongo	**SU**
1	1.026	0.743	1.025	**1.380**
2	1.083	0.755	1.253	**1.659**
3	0.004	0.004	0.260	**70.563**
4	0.004	0.004	0.265	**59.052**
5	0.000	0.000	0.103	**290.828**
6	0.039	0.020	0.237	**11.834**
7	0.039	0.019	0.200	**10.604**
8	0.001	0.001	0.139	**165.204**
9	0.001	0.001	0.148	**291.509**
10	0.114	0.036	0.263	**7.247**
GM				**25.614**
AM				**90.988**

Table 3: Runtimes of 100 thousand long-array NoBench objects