CS 764 Project Final Report

Rashi Jalan
Subasree Venkatsubhramaniyen

REPORT OUTLINE

Introduction

JSON [1] is being used as primary representation of data for most of the web-driven applications. MongoDB has been serving as a document store for JSON objects. Currently, NOSQL systems provide a way to store JSON as different objects can have dynamic schema and hence a predefined schema is not used. However, it would be interesting to study the effect of representing JSON data in RDBMS format.

Argo [2] describes a mapping layer from JSON to tables. Users can interact with the JSON collections via sql queries and the queries are translated to operate on underlying tables. Argo3 uses 3 tables, one each for data types string, numeric and boolean for storing the JSON objects. Each of these 3 tables have 3 columns, objid, keystr, valstr, where objid is a surrogate key, keystr and valstr are the JSON key and values. To handle the nested JSON object, the keystr is prefixed with the complete parent keys, separated by '.' or '[]' for nested object, array respectively. The Argo system had been experimented on PostgreSQL and MySQL and the performance of NoBench queries (benchmark defined in [2]) has been compared against MongoDB.

Our project is an extension of the same idea where we study the system on top of Quickstep [3] and compare the performance of NoBench queries on PostgreSQL, MongoDb and Quickstep. We plan to study the effect of various Quickstep parameters, namely Storage format (Row/Column), indices, SMA and block size on the NoBench queries.

Experiment

The experiment is to execute the 12 NoBench queries for the 3 systems namely PostgreSQL, MongoDB and Quickstep and record the time taken by each of the queries. We executed 10 runs of each query and reported the average of 10 runs after chopping off the outliers. We conducted the experiments on cloudlab machines using the Quickstep Bare Metal Box profile which has 10 cores with 2.6 GHz processor speed, 160 GB RAM, 80GB disk space. Our experiments make use of PostgreSQL 9.3.15 and MongoDB 3.2.

Dataset for the experiments were generated using NoBench data generation module as done in [2]. Each of the JSON objects have a pair of unique strings, a unique integer, a boolean value, 2 dynamically typed values, a nested array, a nested object, a series of 10 sparsely populated attributes, an integer in the range 0-999.

The observations are with respect to 1 million and 16 million JSON objects. The parameters for the queries are generated randomly for each of the queries according the selectivity desired by the query. (Eg. query3 requires selectivity of about 0.1% of JSON objects). We have recorded

the observations with and without creating indices on the systems. Also, we have rewritten queries in Quickstep as it does not support Union operation. Also, the SQL queries on PostgreSQL and Quickstep were as produced by the translation logic of Argo mapping layer. For Query 11, which involves a self-join, we have written a map-reduce job in MongoDB.

Index Creation

We created indices on all three systems. In PostgreSQL, we created B-Tree indices on objid, keystr on str, num and bool tables, valstr and valnum on str and num tables. In MongoDB, we created B-Tree indices on str1, num, dyn1 and nested_arr. In Quickstep, we created csbtree (cache-sensitive B+-tree) indices on objid in str, num, bool tables and on valnum in num table. We created sma (small materialized aggregates) indices on objid in str, num, bool tables and on valnum in num table.

NoBench Queries:

Query 1 - This query projects the two common attributes str1 and num from all the objects in the dataset. This query simply tests the ability of the system to project a small subset of attributes that are always present in each object.

Query 2 - This query projects the two nested attributes nested -obj.str and nested obj.num from the entire dataset. Comparing to Query1 exposes the difference in projecting nested attributes.

Query 3 - This query projects two sparse attributes from one of the 100 clusters. This query tests the performance when projecting attributes that are defined in only a small subset of objects.

Query 4 - This query projects two sparse attributes from two different clusters. This query retrieves the same number of attribute values as Query 3, but fetches them from twice as many objects (i.e. the cardinality of the result of Query 4 is twice that of Query 3).

Query 5 - This query selects a single object using an match predicate on str1. This query tests the ability of the system to fetch a single whole object by an exact identifier.

Query 6 - This query selects 0.1% of the objects in the collection via a numeric range predicate on num. This query tests the speed of predicate evaluation on numeric values and the reconstruction of many matching objects.

Query 7 - This query selects 0.1% of the objects in the collection via a numeric predicate that selects values of dyn1 in a particular range. Comparing this query with Query 6 exposes the difference in evaluating a predicate on an attribute that is statically-typed and an attribute that is dynamically-typed.

Query 8 - This query selects approximately 0.1% of the objects in the collection by matching a string in the embedded array nestedarr. This query tests the speed of evaluating a predicate that matches one of several values inside a nested array, and simulates a keyword-search operation.

Query 9 - This query selects 0.1% of the objects in a collection by matching the value stored in a sparse attribute. This query tests evaluation of a predicate on an attribute that only exists in a small subset of the objects.

Query 10 - This query selects 10% of the objects in the collection (by selecting a range of values for num), and COUNTs them, grouped by thousandth. The result will have 1000 groups, each of whose count is 0.01% of the total number of objects in the collection. This query tests the performance of the COUNT aggregate function with a group-by condition.

Query 11 - This query selects 0.1% of objects in the collection (via a predicate on num) and performs a self-equijoin of the embedded attribute embedded obj.str with str1. This query tests the performance of joins, and simulates following social-graph edges for a set of many users.

Query 12 - This query bulk inserts 0.1% new data

Experiment Results

Below are the results for performance of Argo for data size of 1 million JSON objects.

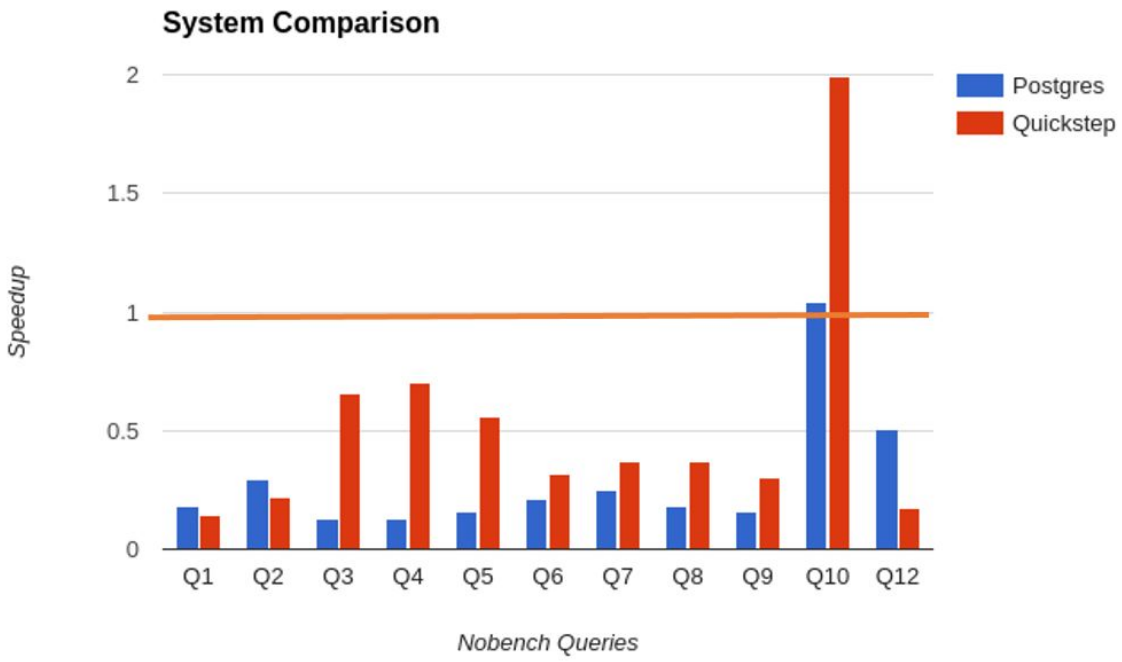

Figure 1: Speedup of Argo without indices

As seen from figure 1,

- **MongoDB seems to perform well** in all queries (as inferred from **speedup < 1**) except for Query 10, Query 11 which can be justified due to map-reduce program in Mongo for computing join
- **Quickstep** seems to perform **better** for most of the queries **when compared to Postgres**

Experiment Without Indices

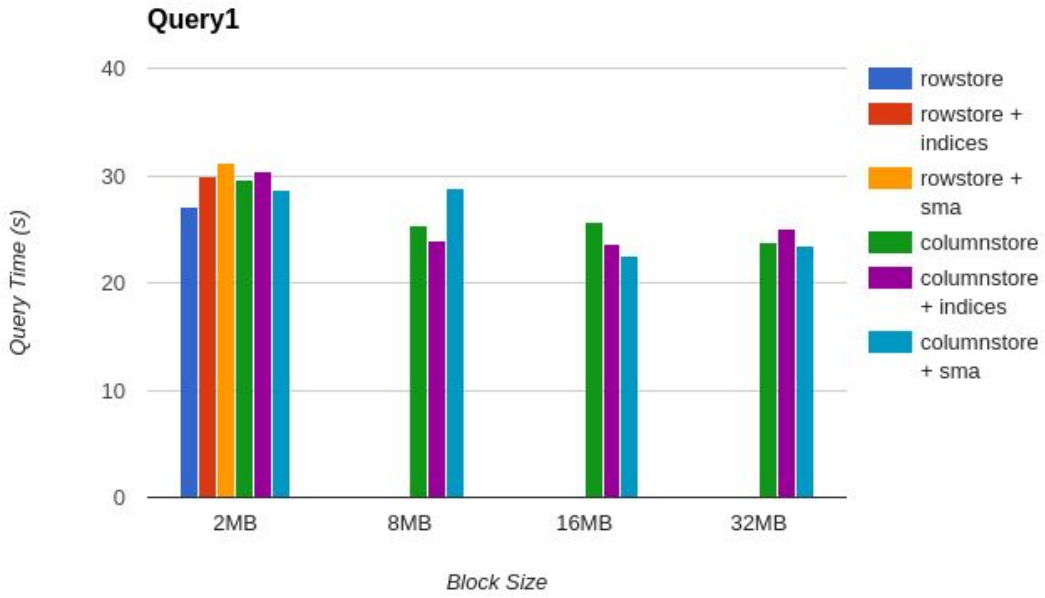

Figure 2: Query 1 performance in Quickstep

```
SELECT str1, num FROM nobench_main;
```

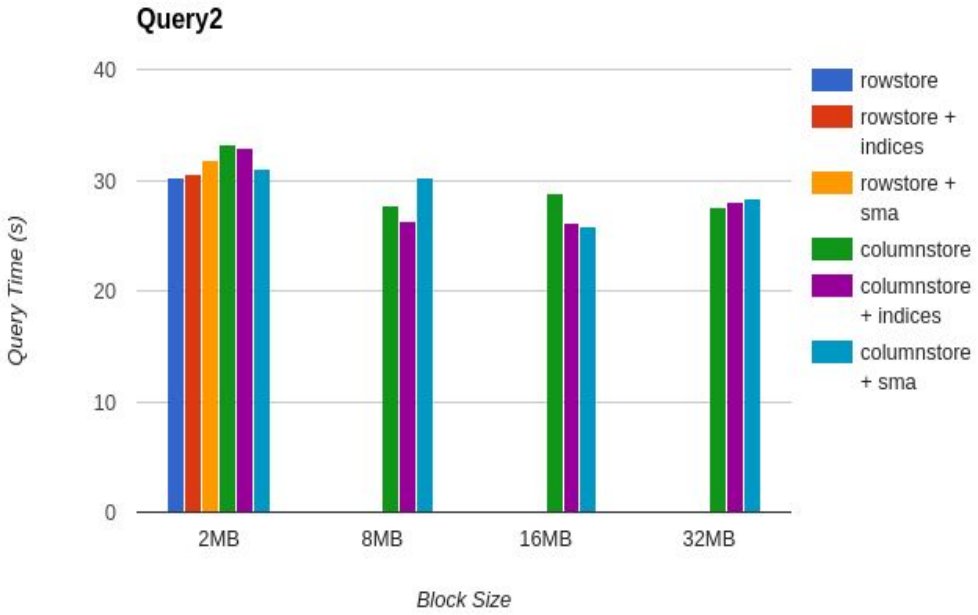

Figure 3: Query 2 performance in Quickstep

```
SELECT nested_obj.str, nested_obj.num FROM nobench_main;
```

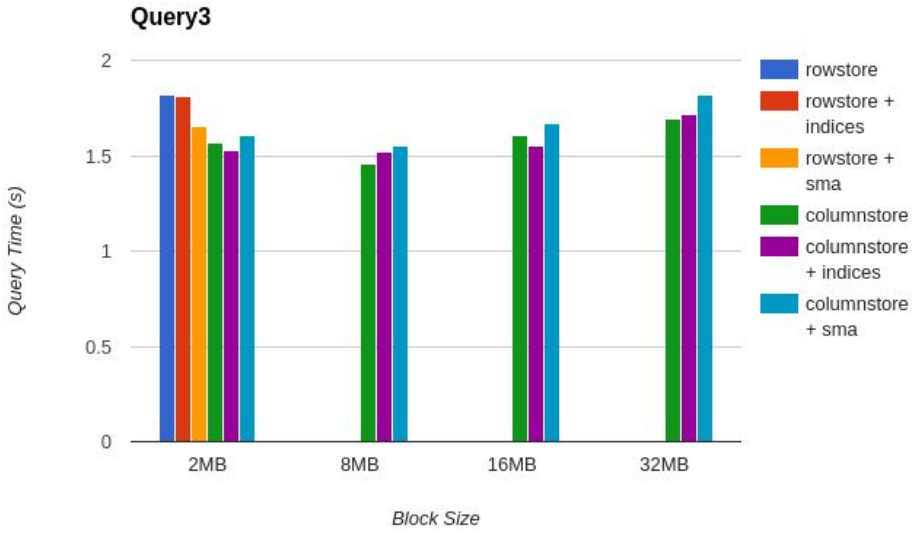

Figure 4: Query 3 performance in Quickstep

```
SELECT sparse_XX0, sparse_XX9 FROM nobench_main;
```

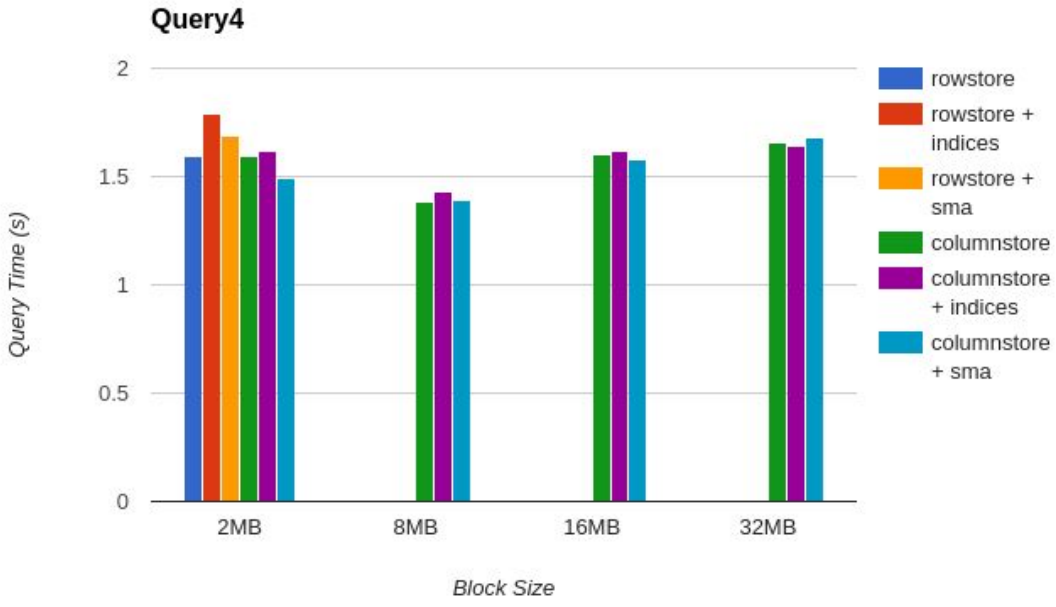

Figure 5: Query 4 performance in Quickstep

```
SELECT sparse_XX0, sparse_YY0 FROM nobench_main;
```

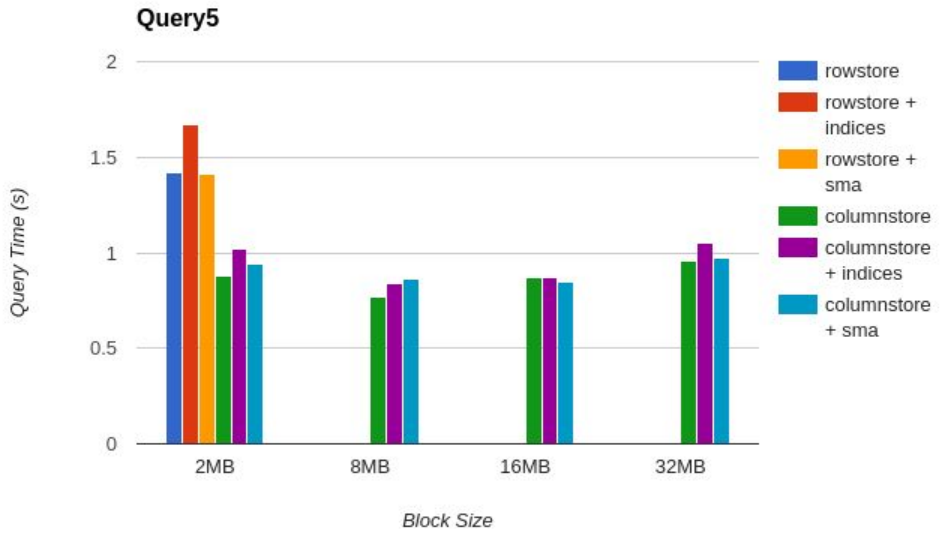

Figure 6: Query 5 performance in Quickstep

```
SELECT * FROM nobench_main WHERE str1 = XXXXX;
```

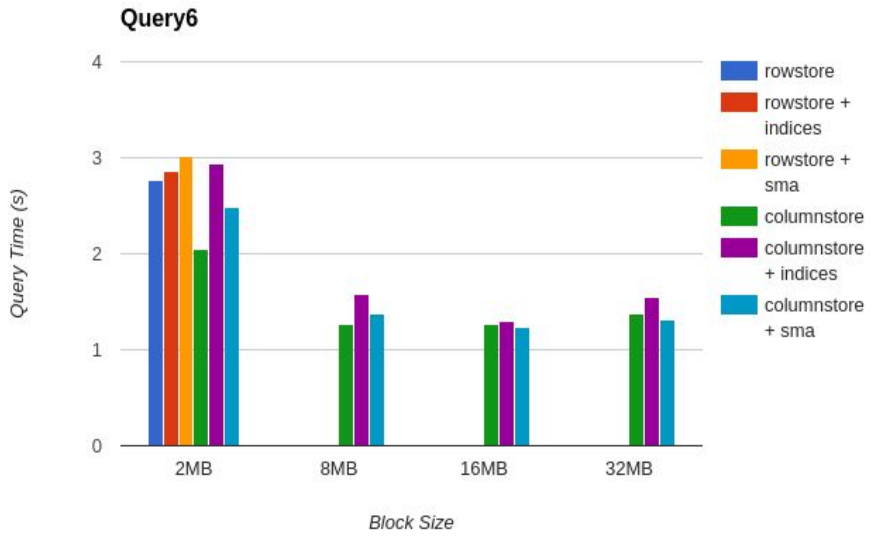

Figure 7: Query 6 performance in Quickstep

```
SELECT * FROM nobench_main WHERE num BETWEEN XXXXX AND YYYYY;
```

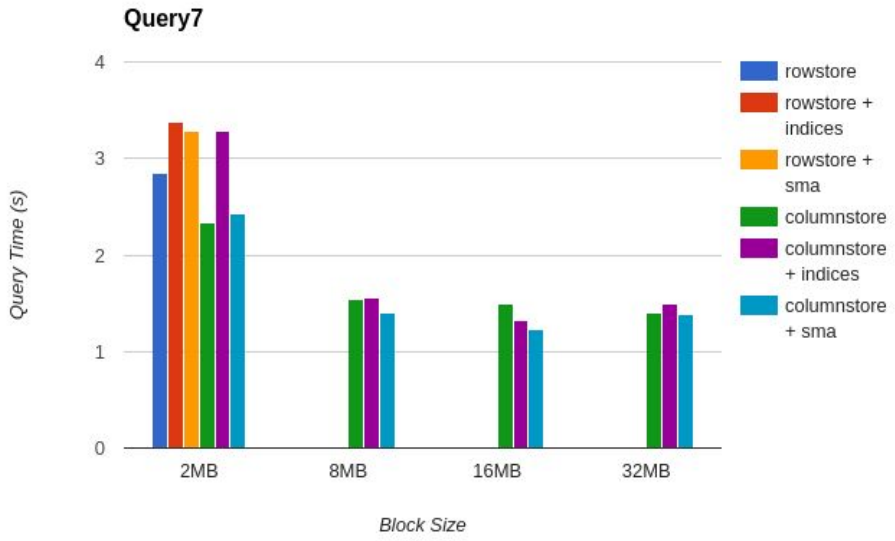

Figure 8: Query 7 performance in Quickstep

```
SELECT * FROM nobench_main WHERE dyn1 BETWEEN XXXXX AND YYYYY;
```

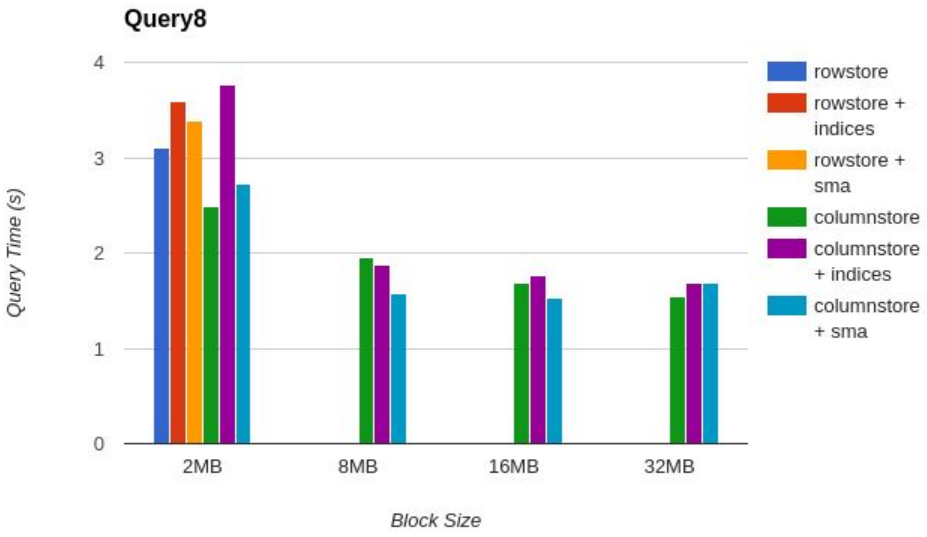

Figure 9: Query 8 performance in Quickstep

```
SELECT * FROM nobench_main WHERE XXXXX = ANY nested_arr;
```

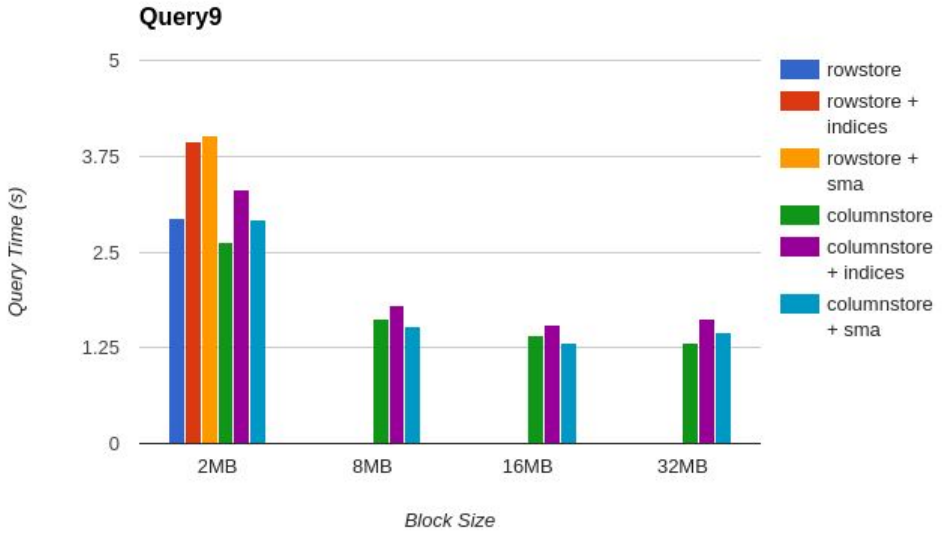

Figure 10: Query 9 performance in Quickstep

```
SELECT * FROM nobench_main WHERE sparse_XXX = YYYYY;
```

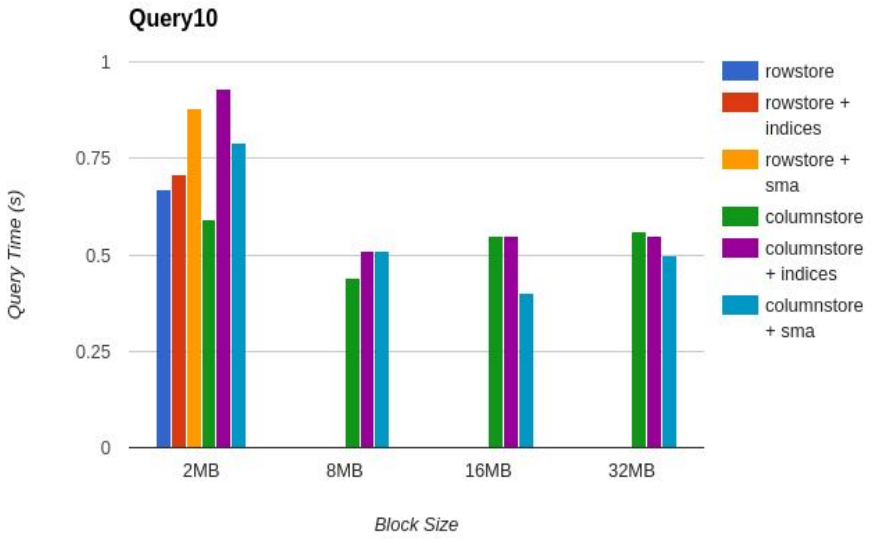

Figure 11: Query 10 performance in Quickstep

```
SELECT COUNT(*) FROM nobench_main WHERE num BETWEEN XXXXX AND
YYYYY GROUP BY thousandth;
```

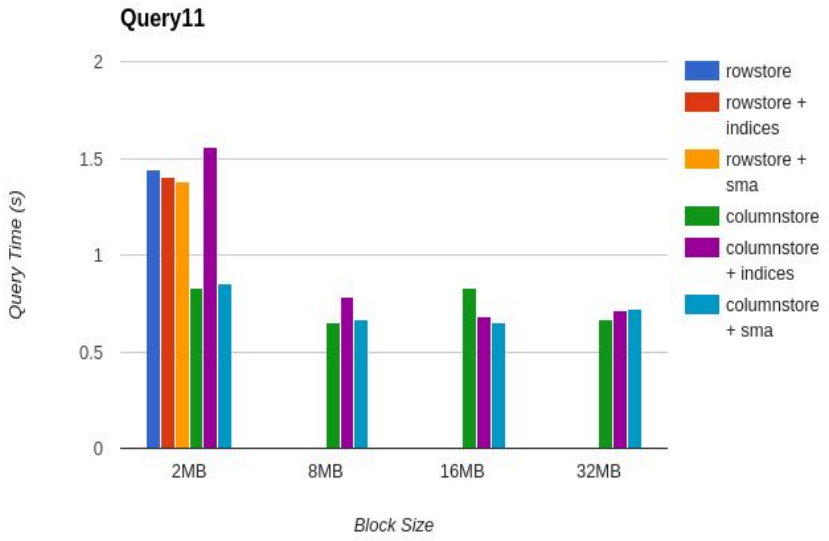

Figure 12: Query 11 performance in Quickstep

```
SELECT * FROM nobench_main AS left INNER JOIN nobench_main AS
right ON (left.nested_obj.str = right.str1) WHERE left.num
BETWEEN XXXXX AND YYYYY;
```

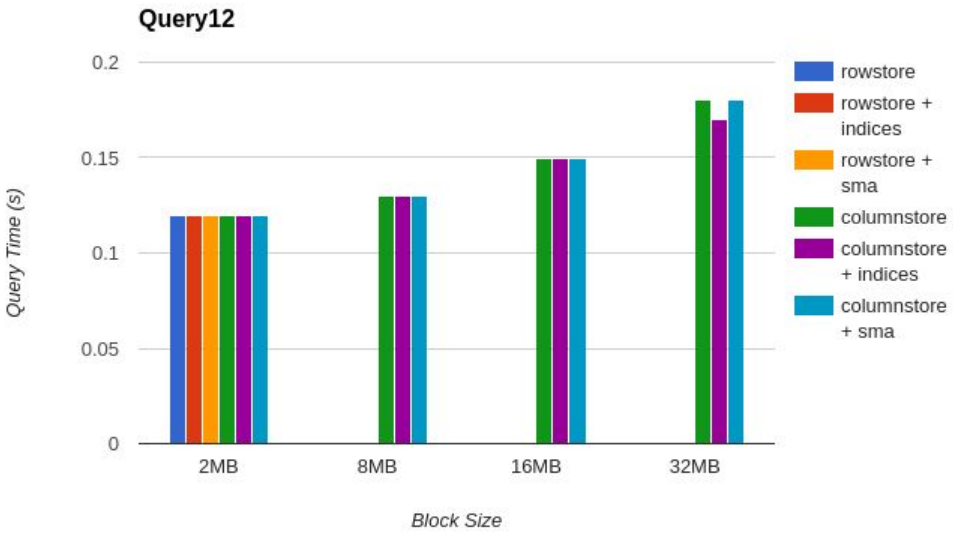

Figure 13: Query 12 performance in Quickstep

```
COPY table FROM file;
```

Experiment With Indices

We ran the experiment by creating indices on MongoDB, PostgreSQL and Quickstep with a block size of 8 MB (optimal size across Query 1 - Query 12) and columnstore. As can be seen from Figure 14, Query 7, Query 9, Query 10, Query 11 perform better in Argo with Postgres than Mongo as opposed to Query 10, Query 11 without indices.

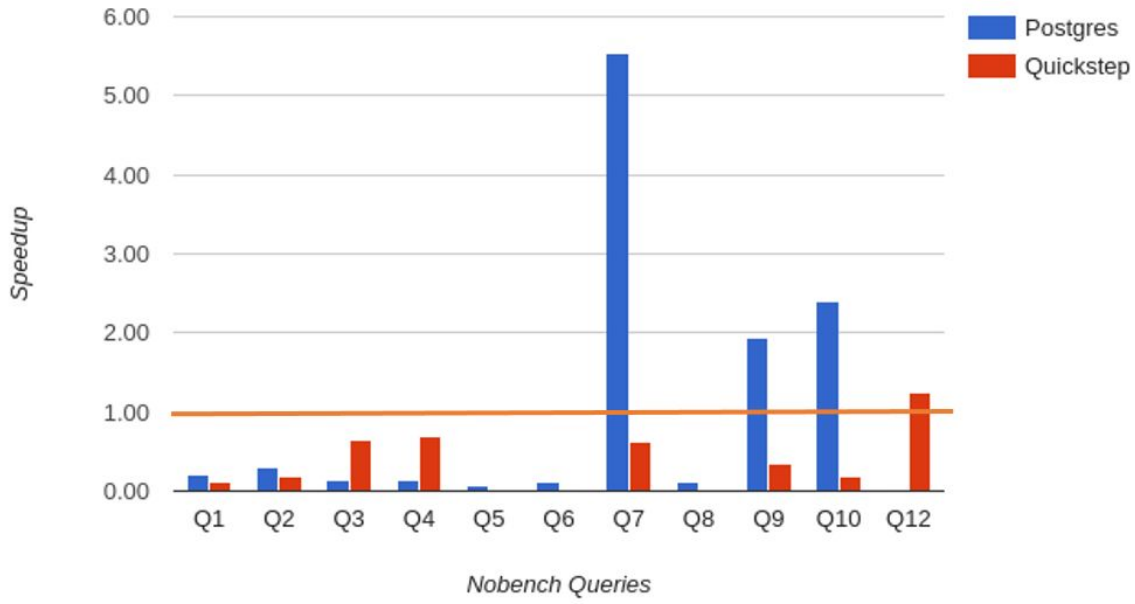

Figure 14: Speedup of Argo with indices

We ran the experiment by rewriting PostgreSQL queries by replacing Union and temporary tables with multiple inserts and making it similar to Quickstep queries. Other parameters are same as previous experiment i.e. indices on MongoDB, PostgreSQL and Quickstep with a block size of 8 MB (optimal size across Query 1-Query 12) and columnstore. As can be seen from Figure 15, PostgreSQL performance for Query 9 and Query 10 went down. Projection queries (Query 1 - Query 4) perform better in Argo with Quickstep than PostgreSQL but all the Selection Queries (Query 5 - Query 9) still performed better in PostgreSQL.

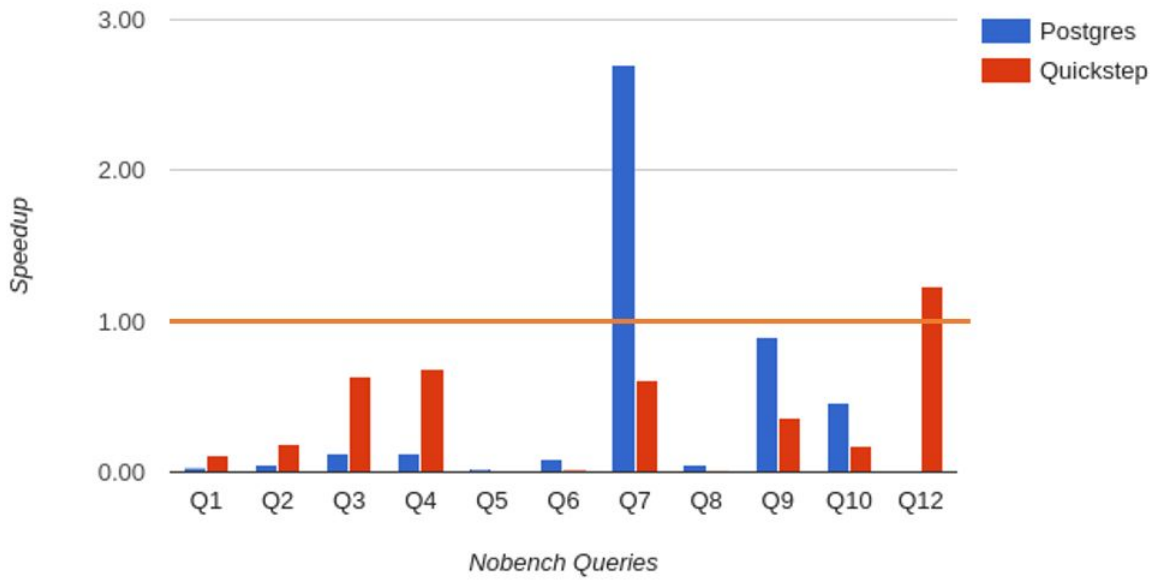

Figure 15: Speedup of Argo with indices and modified PostgreSQL queries

Experiment with 16 Million JSON objects

We also repeated the experiment with data size of 16 million JSON objects and found that MongoDB performed the best as in 1 million experiment. However, Quickstep performed relatively better than PostgreSQL for Query 3, Query 4, Query 10, Query 11, Query 12. Also, PostgreSQL utilized more storage space for indices than Quickstep.

Inference

- **MongoDB performs best** - Argo mapping of JSON to tables could be the bottleneck
- **Join and Aggregate queries** work well in RDBMS
- **Quickstep** performs **better than PostgreSQL** when both systems have **no indices**
- With indices, **PostgreSQL makes better utilization of indices** than Quickstep
- Quickstep
    - **Block size** - 8MB seems to work better than the default quickstep 2MB block size
    - **Storage format** - Compressed-Columnstore works well than rowstore
    - **Sma vs Index** - Sma performed better than Index

Future Work

- Support of Union operator in Quickstep can improve its performance
- Support to create indices for string columns on Quickstep can help improve the performance

Related Work

- Sinew: A SQL System for Multi-Structured Data
  - http://cs-www.cs.yale.edu/homes/dna/papers/sinew-sigmod14.pdf
- Design and Evaluation of Storage Organizations for Read-Optimized Main Memory Databases
  - http://www.vldb.org/pvldb/vol6/p1474-chasseur.pdf
- Enabling JSON Document Stores in Relational Systems
  - http://pages.cs.wisc.edu/~chasseur/argo-long.pdf

References

[1] https://en.wikipedia.org/wiki/JSON
[2] http://pages.cs.wisc.edu/~chasseur/argo-long.pdf
[3] https://quickstep.cs.wisc.edu/