A Comparison of Sinew to Modern Semi-Structured Data Management Systems

Marie Mellor
Rochester Institute of Technology
Rochester, New York, USA
mhm3244@rit.edu

Danny Gardner
Rochester Institute of Technology
Rochester, New York, USA
drg5567@rit.edu

Clinten Hopkins
Rochester Institute of Technology
Rochester, New York, USA
cmh3586@rit.edu

ABSTRACT

Over the last 10 years, the popularity and use of semi-structured databases has exploded. With this, we take a look back at a promising, at the time, solution to the problem of storing semi-structured data that offered PostgreSQL querying, Sinew. Sinew offered a way to be able to store semi-structured data in PostgreSQL using a custom column materializer and "column reservoir" to be able to store both dense and sparse columns in efficient ways. Since then, offerings from MongoDB and PostgreSQL themselves (through PostgreSQL's `jsonb` column type) have had continuous development, offering a more competitive solution to these problems. At the time, Sinew outperformed every offering that the authors compared to, so we decided to re-evaluate the system to see if it still holds up today. Overall, the system seems to perform fairly competitively, though has many implementation issues and could benefit from a re-implementation in modern PostgreSQL.

CCS CONCEPTS

• **Information systems → Database query processing**.

KEYWORDS

Sinew; SQL; Relational Database Management System (RDBMS); NoSQL; JSON; MongoDB; Semi-structured data; Column materialization; Query performance benchmarking; NoSQLBench benchmark suite; Document-oriented database

ACM Reference Format:
Marie Mellor, Danny Gardner, and Clinten Hopkins. 2024. A Comparison of Sinew to Modern Semi-Structured Data Management Systems. In *Proceedings of Adv DB: NoSQL/NewSQL (CSCI-725).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 LITERATURE REPORT

We focused on the comparisons between Sinew and the modern database services MongoDB, and PostgreSQL JSON. Our goal in doing this is to see how Sinew compares to modern database systems in aspects such as batch loading, querying, updating and overall performance and scalability. This section will cover the primary works used in our evaluation.

CSCI-725, Fall, 2024, Rochester, NY

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.1 Sinew

The main objective of Sinew [9] is to enable developers to be able to represent data using self-describing formatting while still utilizing SQL and other traditional database systems. It was built to be a middle layer between multi-structured data and a relational database management structure (RDBMS). Sinew acts as an efficient tool able to convert semi-structured data into something stored and maintained in an RDBMS.

The Sinew system architecture consists of two layers, the use layer and the storage layer. The storage layer allows Sinew to manage and query multi-structured data efficiently. Sinew's schema supports universal relations in which one document corresponds to one row. For every key, value pair from a document, the value will logically get stored in the row and column which corresponds to the document and key. A hybrid approach was used for mapping logical to physical schema to maintain performance benefits, space efficiency and single column serialization. The approach consisted of a combination of "all-physical-column" approach and "all-virtual-column" approach. Using this new hybrid approach, columns for some of the attributes get created while the rest are stored in a special serialized column called the column reservoir. This enables Sinew to maintain the benefits of using physical columns where they're needed while still keeping less frequently accessed keys stored virtually.

Sinew documents the attribute names, types and storage methods to maintain a correct mapping between logical and physical and make system optimizations possible in a catalog. This catalog keeps track of which keys have been observed, the key type information which has been derived from the data, the number of occurrences of each key, if a column is virtual or physical and a 'dirty' flag. The catalog is split into two parts so Sinew can easily identify logical schema and physical schema.

A schema analyzer is also used to allow Sinew to adapt to evolving data models and query patterns. This schema analyzer evaluates the current storage schema in order to determine a distribution of physical and virtual columns. This was done to minimize the overall cost of materialization while still maximizing increasing performance rates.

A column materializer was used to maintain dynamic physical schema by moving data between the column reservoir and physical columns. This materializer was created with the intent of it being a background process that would run only when there are resources available for it within the system. This is where the "dirty column" as previously mentioned comes into play. A dirty column is where some values for a key might exist in the reservoir while others exist in a corresponding physical column. These dirty columns make sure that dirty bits in a catalog are set for that specific column.

Within the use layer, data is loaded in two steps: serialization and insertion. During the serialization step, the loader first parses each document, making sure that its syntax is valid. After the validation, the document is serialized into the proper format. During the serialization step, the loader also aggregates information about the keys within the dataset such as presence, type and sparsity. This information is then added to the catalog. During the insertion step, all the serialized data is placed into the column reservoir and the dirty flag is set to true in the respective columns. The column data is then moved to physical columns once the dirty bit is picked up which creates the newly loaded data. This was put in place so the system components would remain modular.

Due to the nature of the hybrid storage, queries must match the physical schema; this is where a query rewriter comes into play. Here, queries are converted into an abstract syntax tree so that they can be validated and later sent to the storage layer to be executed. If any of the column references cannot be validated, the physical column is rewritten.

1.2 MongoDB

The architecture of MongoDB consists of many features that enable a strong and flexible database. MongoDB is a non-relational document oriented database management system that utilizes a document structure to store its data [5]. Similar to RDBMS tables, MongoDB uses collections to hold data within the database, the collections are able to store documents which have different fields. Collections in MongoDB are what stores the data whereas documents represent a specific record within a collection. Unlike SQL databases like MySQL, a document with a field that might not exist in the database does not cause any errors [7].

As of 2014, MongoDB's default storage engine became WildTiger [7]. WildTiger enabled two concurrent writes to update different documents in the same collection without the need for serialization. This feature that was not possible with the original release of MongoDB. The Binary-JSON (BSON) objects in WildTiger are compressed and stored in a hidden index where BSON pairs stores any recordIDs. This increases the performance greatly by reducing the number of I/O transactions.

The process by which MongoDB scales its data is necessary in order to meet any growing demands of an application or database. MongoDB supports two types of scaling: horizontal and vertical [6]. Horizontal scaling, or scaling down, is done by distributing the data across multiple nodes where each node is responsible for storing portions of the data. This is achieved by sharding, a mechanism that involves partitioning a database into smaller parts called 'shards'. These shards are hosted on a separate server forming what is known as a shard cluster. Vertical scaling, or scaling up, is the act of increasing the capacity of a single server by adding CPU, RAM, or other storage resources. In terms of MongoDB, this would mean upgrading the hardware to something more powerful. This may prove to be effective overall but has its own limitations.

1.3 PostgreSQL JSON

PostgreSQL architecture [2] is a combination of shared memory, background processes, and data files. The shared memory contains a shared buffer, and a WAL buffer. The shared buffer is responsible for minimizing disk input/output so long as it is able to minimize contention when accessed by multiple users. The shared buffer needs to be large enough so that it may be accessed quickly. The shared buffer should keep any frequently used blocks as long as possible. The WAL buffer is used to store changes in the database temporarily in a WAL file. The four main process types that Postgres hosts are: Postmaster, background, backend, and client. The Postmaster process creates the backend processes for the client connections and is the overall parent process for the rest of the processes. The background processes all serve different functions and play a major role in database management. The backend process executes queries before returning the results to the user. This task is reliant on local memory.

In terms of scalability, Postgres scaling[1] requires the understanding and proper implementation of PostgreSQL horizontal scaling as well as vertical scaling. PostgreSQL is kept accessible by high availability (HA) regardless of hardware failures or software crashes. This type of HA implementation requires replication, failover, load balancing and continuous monitoring.

2 COMPLETED WORK

2.1 Sinew Implementation

The main task of this project was ensuring that the Sinew code was able to still load and run in the modern day. In the original paper's implementation of Sinew they used Postgres as the underlying database [9]. For this project, we aimed at simply getting the system to run with the version it was created with (9.3.0). The Sinew repository included multiple bash and make files that perform the initial setup, install the database, and perform some basic tests.

The biggest challenge with implementing Sinew was the lack of documentation in the repository. The initial README file provided no information on how to configure, initialize, or run the system. This left us with a daunting task of having to parse through each file in the repository to understand its function and how it impacted the overall system. In addition to no overarching documentation, the plentiful bash files in the repository were similarly left undocumented, which forced us to analyze each file to understand each one's intended purpose and the order in which to execute them. As we analyzed the original source code we attempted to remedy this situation, creating our own documentation for the repository which will allow future users to reproduce our implementation easily.

We updated the initialization and startup files so that the project can be run on a WSL Ubuntu Linux instance. After the changes to these files the necessary software was able to be installed on the instance and build a test database hosted on postgres. Diving into the Sinew code, there seemed to be many missing files in the repository that were necessary in order to implement the system, including configuration files and the original data directories. The document parser, schema analyzer, and the coulumn upgrader were developed as custom Postgres extensions, which required a global configuration file that was missing. Thankfully, the original developers created options in their makefiles to utilize PGXS, the Postgres extension makefile format, which is a built-in feature since version 8.0. This framework provided many basic makefile configurations that were feasbile to run on this project. After editing the installation bash file to utilize PGXS, the custom extensions were able to

be installed. In addition to the installation script, we made slight modifications to the premade run script that allowed the postgres server to function, allowing Sinew to run.

As for the Sinew source code, limited modifications were needed to ensure the system ran as intended. These extension files were developed in C to implement the necessary functions that Sinew needed. Upon the inital tests, the repository was having segmentation faults due to a buffer overflow occuring during the document serialization. A minor fix was implemented to correct this error and allowed for documents to be properly serialized and loaded into the database. While the main memory issue was resolved to allow Sinew to function properly, further analysis showed many other memory leaks and issues within the repository. These errors did not compromise the system completely, but as shown later in the results it prevented Sinew from operating as fully as it once had. It was not determined whether these issues had been there from the start or were the result of outdated C code.

To load data into the Sinew database, there was a bash script that extracted the data files and exectued stored SQL queries that invoked the serialization function. This would store all loaded data as serialized documents in the column reservoir. This script was modified to locate the directory where our new dataset was stored and perform the SQL load queries. These loading queries also needed slight modifications in order to locate our new data. After these changes were made the system was able to effectively load the database.

Similarly to the benchmark loading functions, the system test files were outdated and needed to be updated in order to effectively test the benchmark queries. The bash script was updated in order to execute the given query files and store their runtime in an output directory. Additionally, we added the ability to load and clear the database from this query to simplify the testing process. This testing script also performed the upgrades that created physical columns from the reservoir that allowed for the SQL functions to be exected on the table. This upgrade function was in the form of SQL queries that utilized user-defined functions provided by the original developers. This file was modified to use the correct data table that the queries were executed on. Finally, the query SQL scripts required some syntax updates to streamline the testing process. After these modifications, Sinew was able to perform the necessary tasks needed for this project.

2.2 MongoDB Benchmark

In the original paper the authors compared the implementation of Sinew against MongoDB and Postgres JSON. During this phase, we also started configuring the MongoDB benchmark system. We will expand on that below. During these experiments the database systems were evaluated on their performance of varying queries from the NoBench NoSQL benchmark suite [8]. Testing on Mongo was pretty straightforward in terms of implementation. [8] has pre-written queries for Mongo version 2.0.0, but as the program has evolved, some of those queries were out-of-date. The most obvious being Queries 10 and 11; for those queries, we had to re-implement the queries with Mongo's `aggregate` framework, released in Mongo 2.2, with support for `$lookup` operations added in Mongo 6.0 [4].

2.3 PostgreSQL JSON Benchmark

As mentioned in the original paper, PostgreSQL 9.3 was one of the systems that Sinew was compared to in terms of performance. Through storing the data in `jsonb`, we were able to load all 16 million rows to the database and recreate the queries of the original paper with little to no issue. However, many advancements have been made in the last ten years since the paper was written. For our experiments, we recreated the queries from the paper using a much more modern version of the database, PostgreSQL 14.4. A significant difference between 9.3 and 14.4 is that `jsonb` was not supported at the time of the original paper [10]. This proves to be a significant detail since we stored our data in `jsonb` format within the database.

Due to `jsonb` not being available at the time the paper was originally published, this also meant that `jsonb`-modifying operators and functions had also not been available. Without these advancements, we would not have had access to the operators and functions that enable queries to delete, modify, and insert into `jsonb` values. It is apparent that PostgreSQL 14.4 also has many improvements regarding performance that were not available at the time of the initial paper such as has aggregation being able to use disk as well as parallel merge joins and queries.

2.4 NoSQLBench

In the original paper the authors used the NoBench NoSQL benchmark suite to run their analysis. As this is a closed-source tool, we did not have access to it for our analysis. In lieu of that, we chose to use an existing, open-source, tool that has a similar feature set and worked well for our use case; this tool was NoSQLBench [8].

This tool was built for NoSQL databases like MongoDB, Neo4J, Pulsar, etc. [1], but did not have any support for PostgreSQL JSON or Sinew. This meant we could not use this tool for our entire testing suite. Our solution was to generate all of the required data (per the NoSQL Benchmark's specifications [3]) and then manually run the queries on the environments ourselves. Sinew already had this feature built into the repo [2], so we were able to easily run that (once our refactoring was done). MongoDB was easy enough to implement as we just set up a JavaScript file with all of our queries and used JavaScript's timers to track the time. PostgreSQL was also relatively simple as we were able to open a connection with a Python script and run the queries directly.

3 EXPERIMENTS

Our experiments try to mimic the experiments done in [9] by replicating the queries created in [3]. We ran all of our testing on isolated Ubuntu 24.04.1 instances. The system had a 3.60 GHz, 6-Core, AMD Ryzen 5 3600 processor with 16 GB of memory and 2 TB of NVMe solid-state storage. We executed each of the 11 queries (excluding the update task defined in [9]), at least 4 times and took the average of the results.

3.1 Load Times

The initial paper evaluated the load time and data sizes of all of the evaluated systems; our evaluation only looked at the total load

[1] https://github.com/nosqlbench/nosqlbench/tree/main/nb-adapters
[2] https://github.com/danieltahara/sinew/tree/master/benchmark/system/document

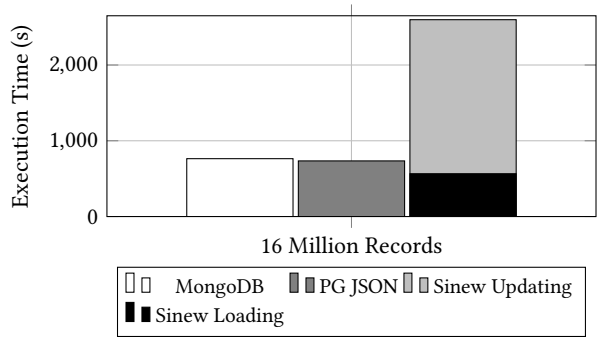

Figure 1: Load Time

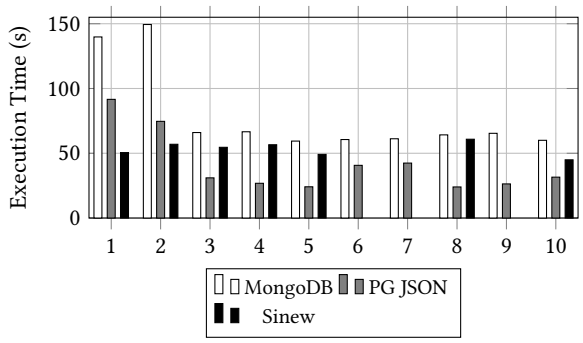

Figure 2: NoSQLBench Query Performance (Q1-Q10)

time. When comparing our results to [9], we noticed an interesting discrepancy in the Sinew load time. Sinew, seemed to perform as well as MongoDB in its initial loading, while only being twice as slow as PG JSON. When we ran our evaluation, we got similar load times for Sinew, actually beating the PG JSON load times, but we noticed there was a secondary operation that took place before the data was actually queryable. This "update" step pushed the total time to over three times longer than both the MongoDB and PG JSON loading. While this is not technically the "loading" of the data, and in theory, this updating could happen in the background while Sinew is running (as described in section 5 of [9]), in the current implementation of Sinew, this is not possible, so the update step is required for any querying to take place. These are noted as separate operations in Figure 1, but we wanted to mention it as it impacts the results of this section greatly.

3.2 Projections

Queries 1 through 4 are simple data projections of top-level, nested, and sparse keys. Figure 2 shows that overall MongoDB did perform the slowest on the data provided. For queries 1 and 2, Sinew performed almost one-third of the time, compared to MongoDB. PostgreSQL followed closely behind Sinew, with its performance increasing for Query 2 (the nested key lookup).

Regarding the sparse queries, Sinew lost its top spot and PostgreSQL took the lead in terms of execution time. PostgreSQL performed in about half the time Sinew and MongoDB took to complete the query, and was the only system to see an improvement when running Query 4 (a sparse key with double the cardinality of Query 3). Below we can see the difference in implementation for each of those queries:
MongoDB:

```
db["main"].find({
  "$or":[
    { "sparse_XX0" : {"$exists" : True} },
    { "sparse_YY0" : {"$exists" : True} }
  ]
}, ["sparse_XX0", "sparse_YY0"])
```

PostgreSQL JSON:

```
SELECT
    data ->> 'sparse_XX0' AS sparse_1,
    data ->> 'sparse_YY0' AS sparse_2
FROM main
WHERE data ? 'sparse_XX0' OR data ? 'sparse_YY0';
```

Sinew:

```
SELECT
    document_get_text(data, 'sparse_XX0'),
    document_get_text(data, 'sparse_YY0')
FROM main;
```

3.3 Selections

Queries 5 through 9 were evaluations on both "rifle-shot" selection (Query 5) and range selection. PostgreSQL maintained its leadership in query 5, finding the matching record in under half the time of the other systems.

Queries 6, through 9 were disappointing when it came to evaluating the Sinew. After spending weeks to get the Sinew implementation[3] running, we ran into memory issues that caused queries 6, 7, and 9 to fail every time. The output seemed to be complaining about missing "chunks" but due to the old version of PostgreSQL, we were struggling to find a usable solution to this issue. We eventually decided that these queries were just unexecutable without further development. Of the results we were able to produce, PostgreSQL was again outperforming MongoDB for every query and the gap between Sinew and MongoDB was closing, with the total time being within half a second.

3.4 Joins and Aggregations

Queries 10 and 11 test the performance of `GROUP BY` and `JOIN` operations in the PostgreSQL and Sinew implementations and the `$group` and `$lookup` operations in MongoDB. Figure 2 shows the performance of Query 10 between all three systems. MongoDB was the slowest, even though Query 10 was the fastest query MongoDB was able to perform (with an average time of exactly one minute). Sinew was about 15 seconds faster than MongoDB and PostgreSQL was the fastest with under 30 seconds. The `jsonb` column type

[3] https://github.com/danieltahara/sinew

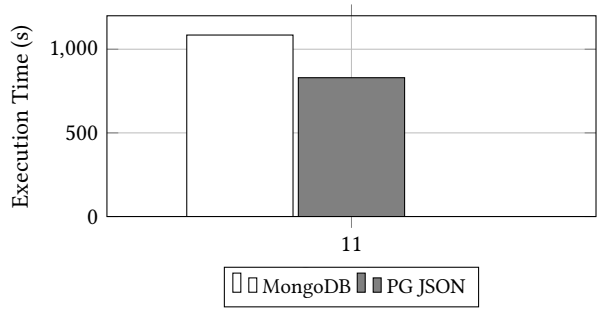

Figure 3: NoSQLBench Query Performance (Q11)

seems to have very high performance, even without indexing enabled.

Query 11's results are shown in Figure 3. The results here were a bit disappointing. Sinew was again struck by the memory issues described in section 3.3. The system was unable to produce any results over any of our executions for this query and would always complain of "missing chunks." Of the results we were able to produce, PostgreSQL outperformed the MongoDB implementation by a little over 15%.

Though it did not complete, we wanted to show the implementation differences for these queries to highlight Sinew's simple syntax.

MongoDB:

```
db["main"].aggregate([
  {
    $match: {
      num: { $gte: XXXXX, $lte: YYYYY }
    }
  },
  {
    $lookup: {
      from: "main",
      localField: "nested_obj.str",
      foreignField: "str1",
      as: "matchedDocs"
    }
  }
]);
```

PostgreSQL JSON:

```
SELECT
    jd.id,
    c.data ->> 'nested_obj' ->> 'str',
    jd.data ->> 'str1' AS str1
FROM main jd
LEFT JOIN main od
    ON jd.data -> 'nested_obj' ->> 'str'
        = od.data ->> 'str1'
WHERE (jd.data ->> 'num')::int >= XXXXX
AND (jd.data ->> 'num')::int <= YYYYY;
```

Sinew:

```
SELECT *
FROM main t1 INNER JOIN main t2
    ON (document_get_text(t1.nested_obj, 'str')
        = t2.str1)
WHERE left.num BETWEEN XXXXX AND YYYYY;
```

REFERENCES

[1] Ahmed, I. Postgres scalability: Navigating horizontal and vertical scalability pathways. Available at https://www.pgedge.com/blog/scaling-postgresql-navigating-horizontal-and-vertical-scalability-pathways.

[2] Bitnine Global. Postgresql architecture, 2024. Available at https://medium.com/agedb/postgresql-architecture-59d6242d91d8.

[3] Chasseur, C., Li, Y., and Patel, J. M. Enabling json document stores in relational systems. *Proc. of WebDB* (2013).

[4] Done, P. *Practical MongoDB Aggregations: The official guide to developing optimal aggregation pipelines with MongoDB 7.0.* Packt Publishing, 2023. Available at https://www.practical-mongodb-aggregations.com/.

[5] GeeksForGeeks. Mongodb architecture, 2024. Available at https://www.geeksforgeeks.org/mongodb-architecture/.

[6] GeeksForGeeks. Scaling in mongodb, 2024. Available at https://www.geeksforgeeks.org/scaling-in-mongodb/.

[7] Nasser, H. Mongodb internal architecure. Available at https://medium.com/@hnasr/mongodb-internal-architecture-9a32f1403d6f.

[8] NoSQLBench Project. Nosqlbench project, 2023. Available at https://docs.nosqlbench.io/.

[9] Tahara, D., Diamond, T., and Abadi, D. J. Sinew: a sql system for multi-structured data. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Jun 2014), 815–826.

[10] The PostgreSQL Global Development Group. Postgressql: Feature matrix. Available at https://www.postgresql.org/about/featurematrix/.