



12/08/2024

Student Name: Matthew Kenneth Peterson

Student ID: 3719754

Student Email: mpeters9@unb.ca

Class: CS 4525 Advanced Databases

Project Title: R2D2:
Rust-Based Relational
Database Design

Supervisor: Dr. Jong-Kyou Kim

DEPARTMENT OF COMPUTER SCIENCE

University of New Brunswick, Saint John E2K 5E2

1 Overview

The goal of this project was to design and implement a time-series database to store and retrieve time-based data efficiently, using our programming language of choice. Specifically, we needed to use a B+ tree to index timestamps for fast insertion, retrieval, and range-based querying. This database was required to support basic CRUD operations and aggregation queries typical in time-series databases, such as retrieving data over a specific time interval.

In order to meet these needs, I decided to use the Rust programming language. Rust is a very strongly-typed, compiled language with a big emphasis on memory management safety. This does however make it very difficult to make list and tree structures with: put simply, its memory ownership system means instead of traditional pointers, it must use more complicated structures for keeping track of data structures' nodes. I originally attempted to implement the B+ tree on my own. This implementation used reference counters and atomic locks as replacements for pointers. However, due to the lack of time necessary to get it working, I instead relied on a pre-existing implementation by André Guedes on GitHub, available as the Rust crate `bplustree`¹. This crate uses a similar rationale, but relies on slightly different data structures to achieve the same goal.

¹<https://docs.rs/bplustree/latest/bplustree/>

2 Implementation

For indexing by timestamps, my database implementation uses unsigned 128-bit integers, which represent the time in milliseconds since the UNIX epoch (January 1st, 1970). The accompanying row-data is stored in a binary-formatted JSON document using the `bson` crate’s² `Document` datatype, which allows for a more flexible database design. That being said, I have also included a separate `bson::Document` which outlines the database’s minimum-required fields and their associated types for all rows in the database; this schema is upheld whenever a key-document pair are inserted into the database.

```
pub struct Database {  
    bptree: Box<GenericBPlusTree<u128, bson::Document, FAN_OUT, FAN_OUT>>,  
    schema: bson::Document,  
    min_timestamp: u128,  
    max_timestamp: u128,  
}
```

Figure 1: Encapsulating data structure for the database approach. The first `FAN_OUT` is the maximum size of internal nodes; the second is for leaf nodes.

Due to my experience implementing a basic HTTP web server using rust for CS3893 (Networking), I used a similar strategy for the database’s interface. The database listens over TCP port 6969 (selected due to its lack of any modern dedicated use), and will respond to any connections made to it, expecting HTTP headers. In order to access the database for any type of query, one must make a GET request to the server over HTTP. Any method for making GET requests can work, however visuals are provided through HTML alone, and so a web browser is recommended. Queries are formatted as

`<ip address>:6969/<operation>[::<category>]::<options>[::HIDE]`

and are case-, whitespace- and character-sensitive. They are detailed below in the API Section.

To set up the database, one would need to modify the main method it is run from, as there is no API functionality for creating or destroying tables. The code by default assumes data as formatted in Samuel Cortinhas’s ”Time Series Practice Dataset” on kaggle³, and this same dataset will be used in the Benchmarks Section to compare the database’s performance with Python’s SQLite implementation.

²<https://docs.rs/bson/latest/bson/>

³<https://www.kaggle.com/datasets/samueltcortinhas/time-series-practice-dataset?resource=download&select=train.csv>

```
let mut database : Database = Database::new(  
    fields: vec![  
        String::from(s: "store"),  
        String::from(s: "product"),  
        String::from(s: "number_sold")  
    ],  
    types: vec![  
        String::from(s: "number"),  
        String::from(s: "number"),  
        String::from(s: "number")  
    ]  
);
```

Figure 2: The constructor used in the main method to initialize the database.

3 API

3.1 Data Retrieval: LIST

Returns an HTML table with the results of the LIST query. Similar to a SELECT query in SQL, but far more limited.

If no options are given to the LIST query, the database's metadata will be returned (equivalent to `LIST::METADATA`).

Appending `::HIDE` to the end of a LIST query will not return the full HTML table to the user, but rather will only return a status message. This is specifically useful for gathering performance metrics, as transferring larger HTML files creates unnecessary network overhead for simple benchmarks.

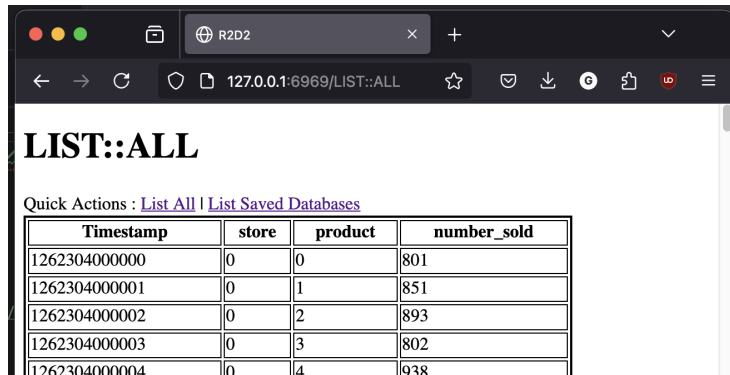
3.1.1 Usage

```
LIST::
```

Examples:

View all rows in the database:

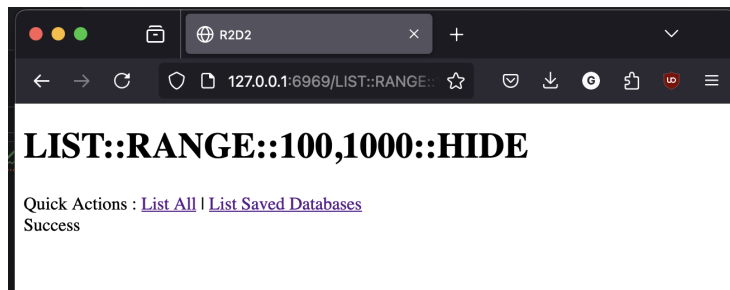
```
LIST::ALL
```



Timestamp	store	product	number_sold
1262304000000	0	0	801
1262304000001	0	1	851
1262304000002	0	2	893
1262304000003	0	3	802
1262304000004	0	4	938

Benchmark the speed of accessing all rows with keys in range [100,1000]:

```
LIST::RANGE::<100,1000>::HIDE
```



Timestamp	store	product	number_sold
1262304000000	0	0	801
1262304000001	0	1	851
1262304000002	0	2	893
1262304000003	0	3	802
1262304000004	0	4	938

3.1.2 Categories

- **ALL**
Lists all rows in the database.
- **ONE::<timestamp>**
Lists a single row of the database whose key matches the provided timestamp, if any.
- **RANGE::<timestamp A>,<timestamp B>**
Lists all rows in the database whose keys fall between A and B, inclusive (so [A,B]).
- **METADATA**
Retrieves the metadata about the database. Currently unused and only holds a few values, but could be useful in the future.
- **SAVED**
Lists all saved databases in the **data** folder. This folder is created at runtime if it doesn't exist in the directory from which R2D2 is run.

3.2 Data Aggregation: AGGREGATE

Returns an HTML table cell containing the result of an aggregation over the whole of a single column in the database. This only works on columns of type `number`, as any values found that aren't this type will be treated as a 0.

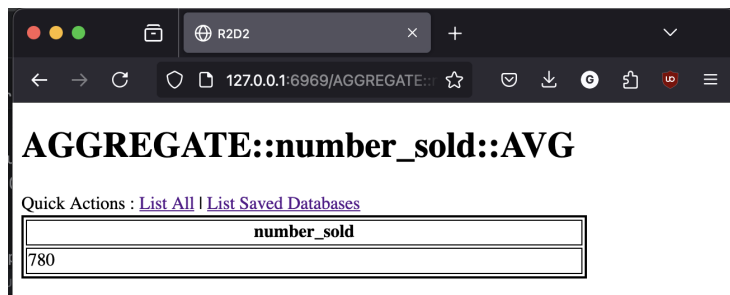
3.2.1 Usage

`AGGREGATE::(<category>)`

Examples:

View the average value of the column `number_sold`:

`AGGREGATE::number_sold::AVG`

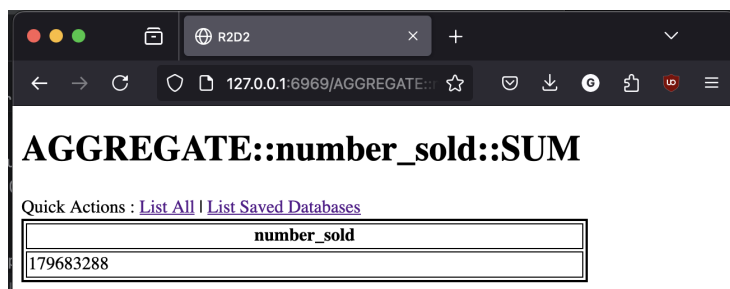


The screenshot shows a web browser window with the address bar displaying `127.0.0.1:6969/AGGREGATE::`. The main content area displays the title **AGGREGATE::number_sold::AVG**. Below the title, there are quick action links: [List All](#) and [List Saved Databases](#). A table is displayed with the following structure:

number_sold
780

View the sum of the column `number_sold`:

`AGGREGATE::number_sold::SUM`



The screenshot shows a web browser window with the address bar displaying `127.0.0.1:6969/AGGREGATE::`. The main content area displays the title **AGGREGATE::number_sold::SUM**. Below the title, there are quick action links: [List All](#) and [List Saved Databases](#). A table is displayed with the following structure:

number_sold
179683288

3.2.2 Categories

- SUM
- AVG
- MIN
- MAX

3.3 Data Insertion: INSERT

Inserts a row into the database using provided column-value pairs. Any column can be left empty. If no valid columns are specified, or any one invalid column is specified, the operation will fail and the database will notify the user. When an insert fails, no changes will be made to the database. When an insert succeeds, the database will present the new database to the user (equivalent to a `LIST::ALL`), unless `::HIDE` is specified.

A timestamp may be provided with the database. If not, the database will automatically use the time of the data's insertion as its timestamp.

If a timestamp already exists, whether or not it is user-specified, a collision will occur. Collisions are automatically handled by the database, as upon detecting a collision, the database will re-attempt the insertion by incrementing the timestamp by 1. The database will re-attempt as many insertions as needed until the insert operation succeeds. As timestamps are formatted as time in milliseconds since the UNIX epoch (January 1st, 1970), the consequences of this increment are assumed to be negligible.

3.3.1 Usage

```
INSERT
  ::<column>=<value>{,<column>=<value>}
  [::TIMESTAMP=<timestamp>]
  [::HIDE]
```

Example:

```
INSERT
  ::store="Walmart",product_id=101,available=false
  ::TIMESTAMP=1733697225084
  ::HIDE
```

3.4 Data Deletion: REMOVE

Removes a single database entry, by timestamp. More options are planned, but have not yet been implemented. Currently, executing any `REMOVE` query other than `REMOVE::ONE` will return an error.

3.4.1 Usage

```
REMOVE::ONE::TIMESTAMP=<timestamp>
```


3.5 Data Serialization: SAVE

Collects the database's contents into a serialized format for later retrieval. Supports two serialization types: a special form of JSON, using the `.r2d2` file extension; and a plain comma-separated value (CSV, extension `.csv`) representation. All serialized data will be put in the `data` folder. This folder is created at runtime if it doesn't exist in the directory from which R2D2 is run.

The JSON format is preferred. While it takes up significantly more disk space (see more in Metrics section), it loads much quicker, as it is the result of the included serialization by the `bson` library's `Document` type, which is how row-data is saved in the database.

The CSV format is only used for exporting the database's data for use by other programs, and reloading CSVs into R2D2 is not supported at this time.

When serializing into JSON, a filename must be provided by the user. The extension `.r2d2` will automatically be applied to the resulting JSON. The filename will also be sanitized, eliminating the possibility of file-system injection attacks. If the file specified already exists, the database will attempt to append `_1.r2d2` instead, and will increment this number until the filename does not exist already. On a successful serialization, the list of all saved databases will be shown (equivalent to `LIST::SAVED`).

When serializing into CSV, a file named `dump.csv` will be created in the `data` folder mentioned previously. This will overwrite any previous version of `dump.csv`. A new name cannot be specified.

3.5.1 Usage

```
SAVE::  
  ::NAME=<filename>  
  ::CSV
```

3.6 Data De-serialization: LOAD

Attempts to de-serialize a given `.r2d2` file into the database. This will overwrite all data currently held in the database, and cannot be undone.

If the load operation succeeds, the contents of the database will be displayed (equivalent to `LIST::ALL`). If the load operation fails, a notice will be shown to the user. If the

3.6.1 Usage

```
LOAD::<filename>
```

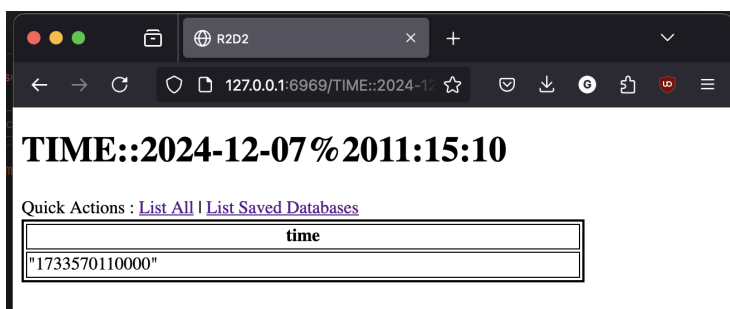
3.7 Current Timestamp: TIME

Will return the UNIX timestamp in milliseconds since January 1st, 1970, for a given date and time. The provided date and time must be in the ISO 8601 form of YYYY-MM-DD HH:MM:SS. Note that the timezone is UTC.

3.7.1 Usage

TIME::YYYY-MM-DD HH:MM:SS

Example TIME::2024-12-07 11:15:10 will produce 1733570110000.



4 Benchmarks

4.1 Timing

In order to test the timed performance of the database implementation, a Python script was created which tests and compares the database functionality with Python’s implementation of SQLite. This script can be found in the GitHub repository, under `srccsv_to_r2d2.py`. Using a list of specified test cases, which limit the database to a size n , the script does the following:

1. Loads the testing dataset (`'train.csv'`) into a Pandas dataframe, with maximum size of n .
2. Inserts up to n lines of the dataframe into the custom database, recording the time it takes to insert every batch of m lines.
3. Inserts up to n lines of the dataframe into the SQLite database, recording the time it takes to insert every batch of m lines.
4. Randomly shuffles the shortened dataframe of n rows.
5. Queries all n entries inside of either database in this same shuffled order, recording the time it takes to insert every batch of m lines for each.
6. Randomly generates r ranges of timestamps to query from either database, recording the time it takes to collect each range.
7. Removes all n lines from either database, recording the time it takes to remove every batch of m lines.

The values of m , and r are dependant on the size of each subset of the test dataset, n ;

$$m = \begin{cases} 1000 : & n > 10000 \\ \frac{n}{100} : & n \leq 10000 \end{cases}$$
$$r = \begin{cases} \frac{n}{100} : & n > 1000 \\ 100 : & n \leq 1000 \end{cases}$$

In turn, this creates the test cases as represented in Figure 3. The final results for each test case can be found below, in Figures 4 through 8. Note the excellent and consistent performance of range queries as the datasets get larger; this is a feature of using a B+ tree.

Test Subset Size (n)	Line Batch Size (m)	# of Range Queries (r)
100	1	100
1000	10	100
10000	100	100
100000	1000	1000
230090	1000	2300

Figure 3: The test cases generated from the conditions of m and r using the chosen values for n . Note that 230090 was the full size of the dataset.

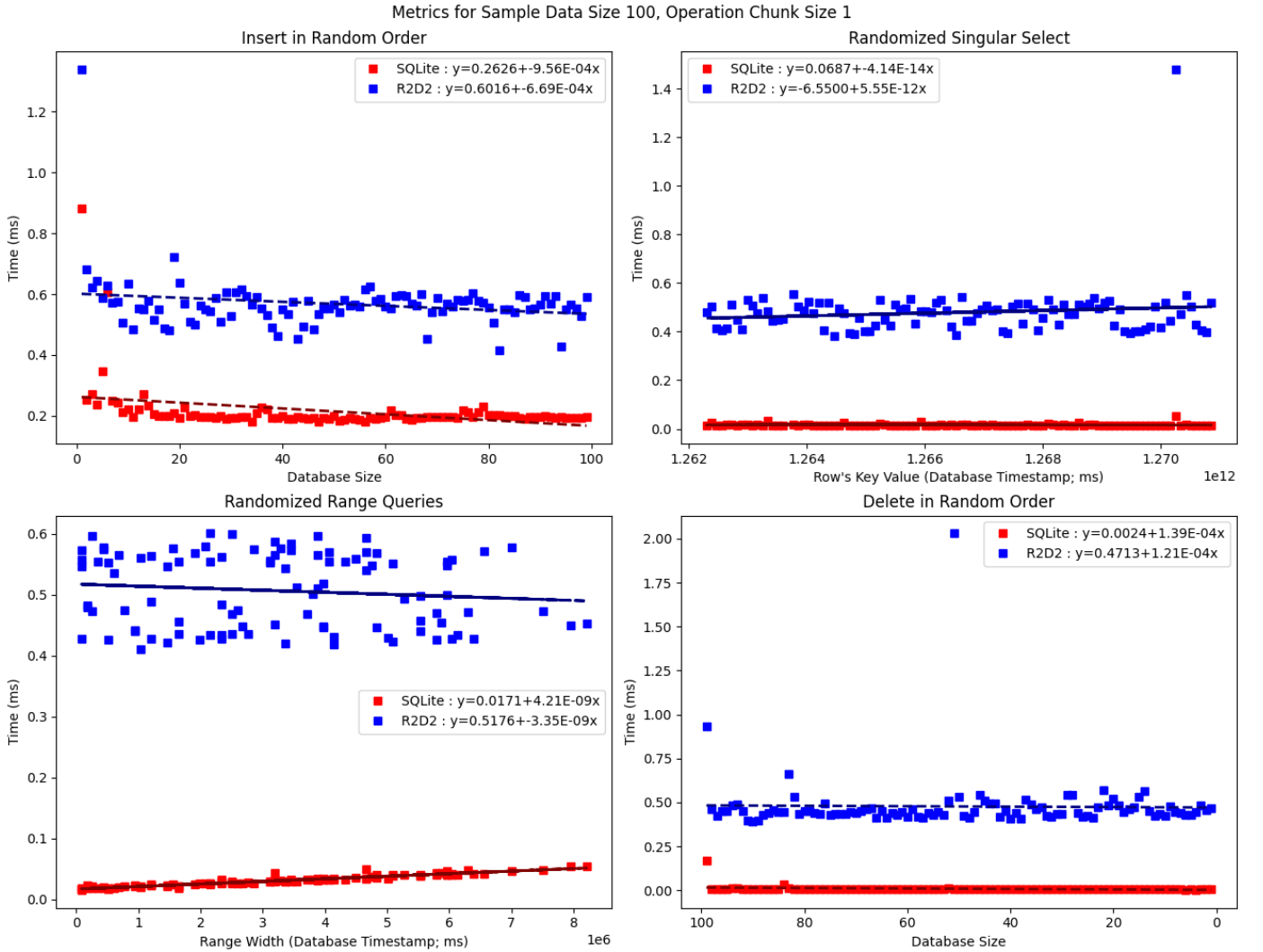


Figure 4: $n = 100, m = 1, r = 100$

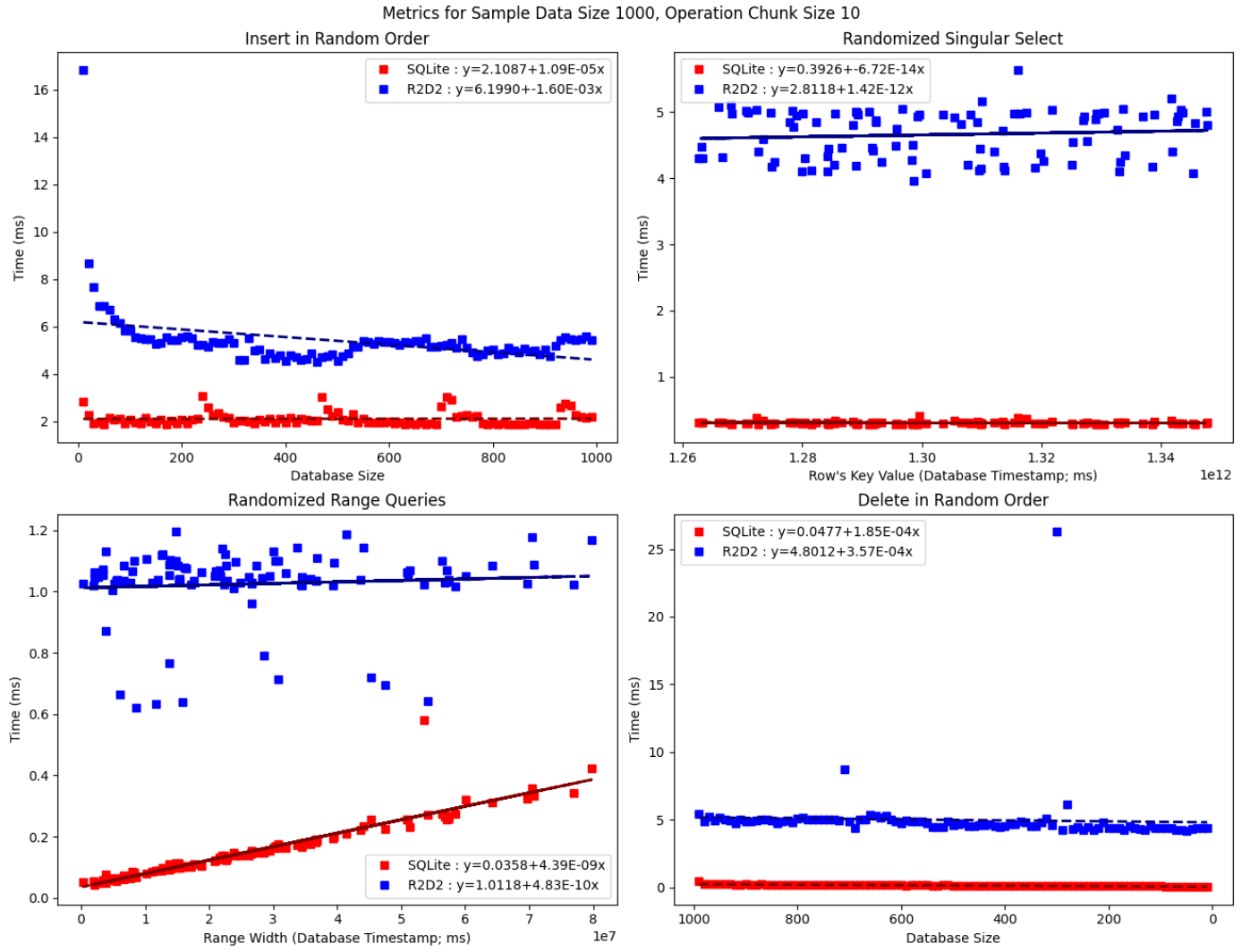


Figure 5: $n = 1000, m = 10, r = 100$

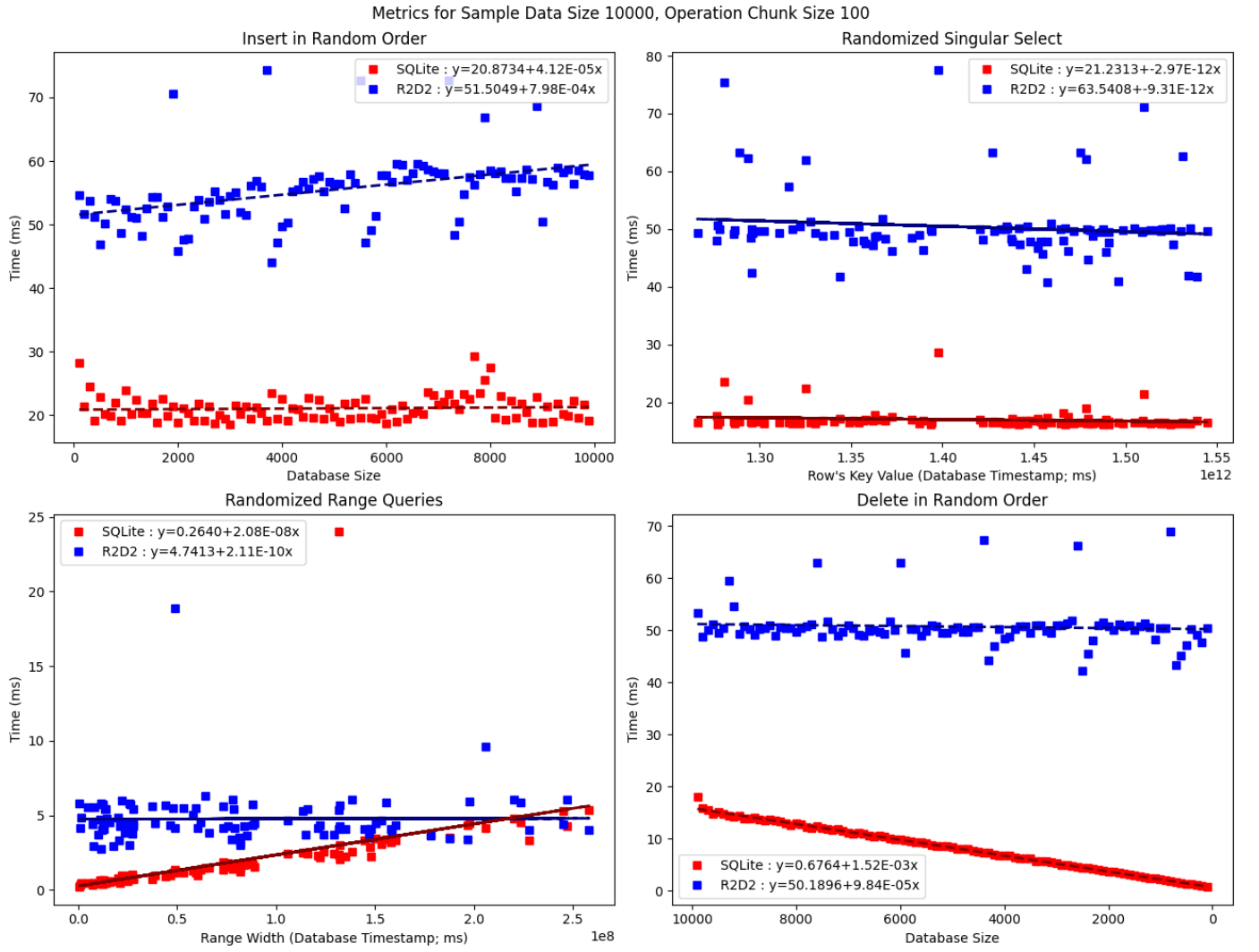


Figure 6: $n = 10000, m = 100, r = 100$

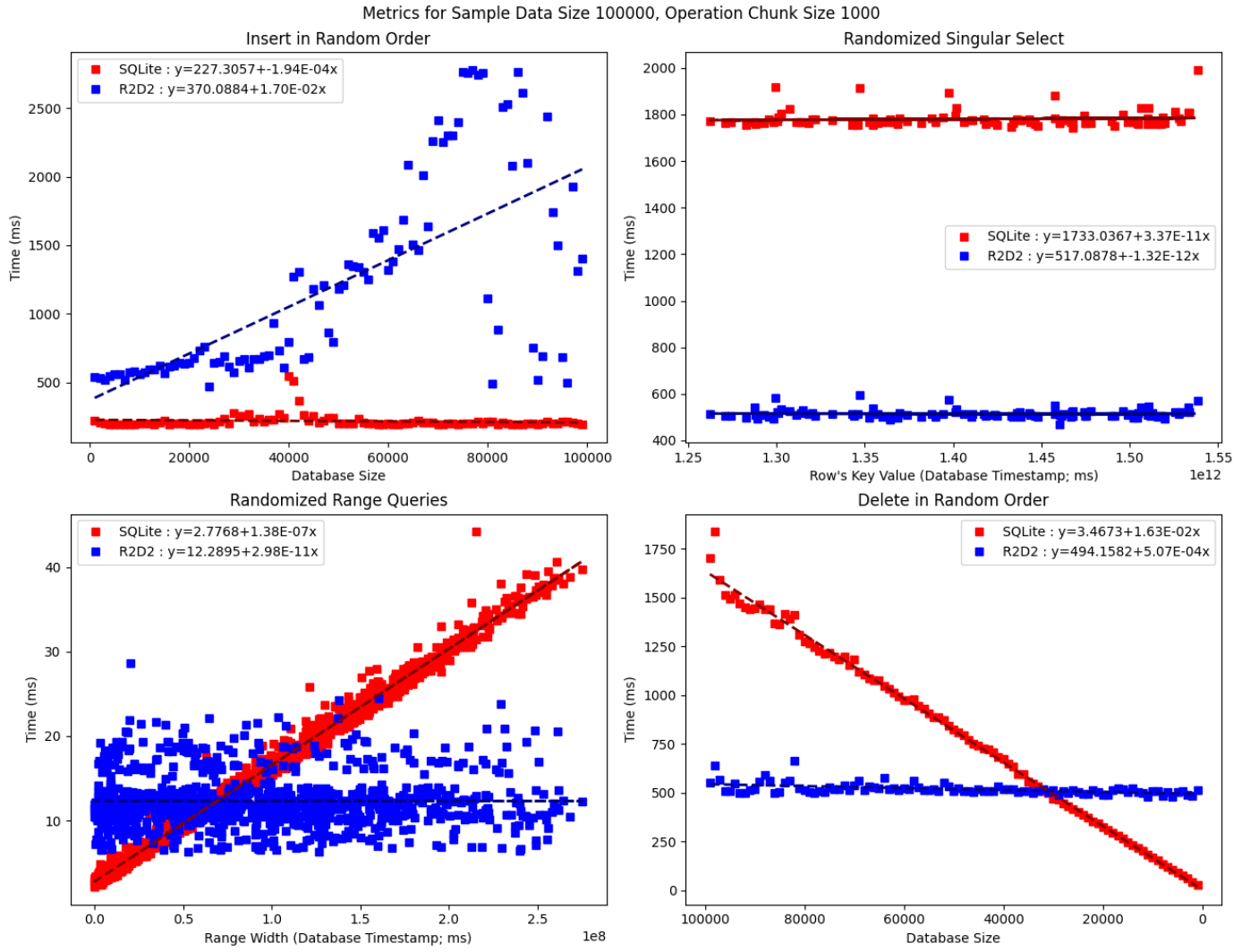


Figure 7: $n = 100000, m = 1000, r = 1000$

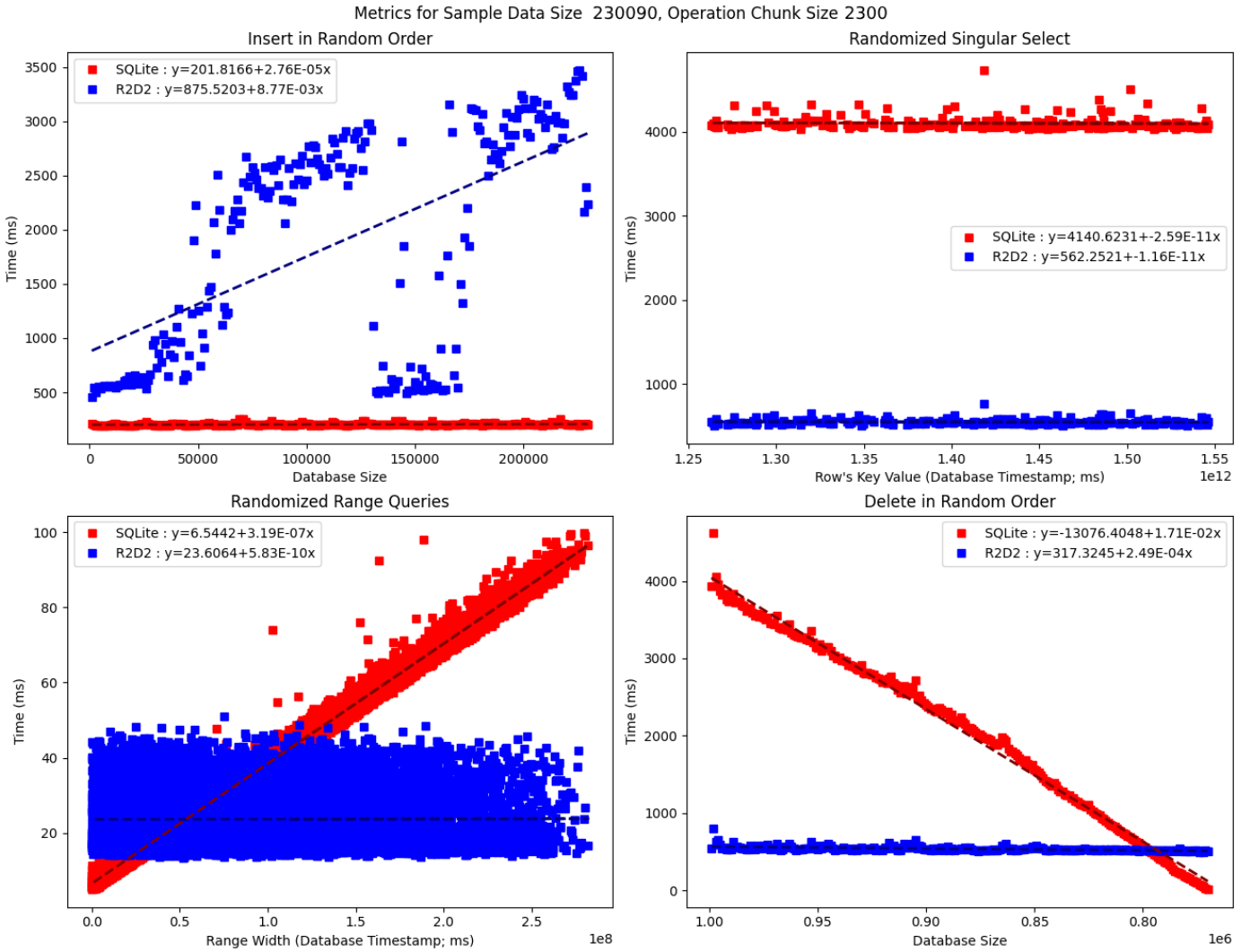


Figure 8: $n = 230090, m = 1000, r = 2300$. It looks a little crazy, but it gets the point across.

4.2 File Sizes

While the R2D2 database seems to excel at query performance as the size of the data store grows, it also suffers in terms of the resulting file size. By storing each row as a separate JSON structure during serialization, the database also ends up storing a redundant column label for every field stored in a row, lots of whitespace and lots of JSON brackets. Over time, this adds up, and this creates the difference in final file sizes as seen in Figures 9 and 10.

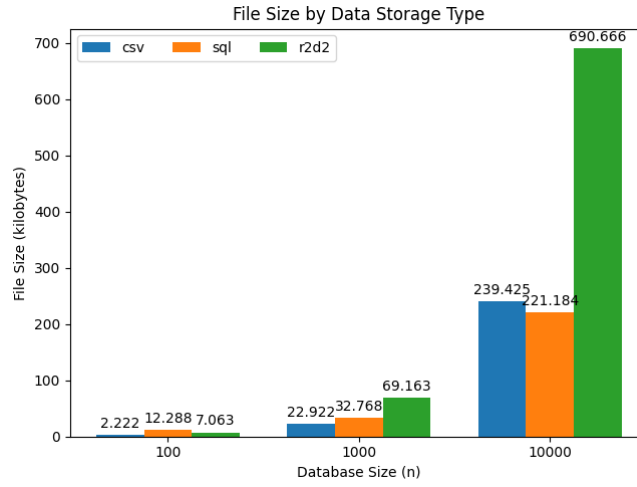


Figure 9: File sizes in **kilobytes** for $n=100$, $n=1000$, and $n=10000$.

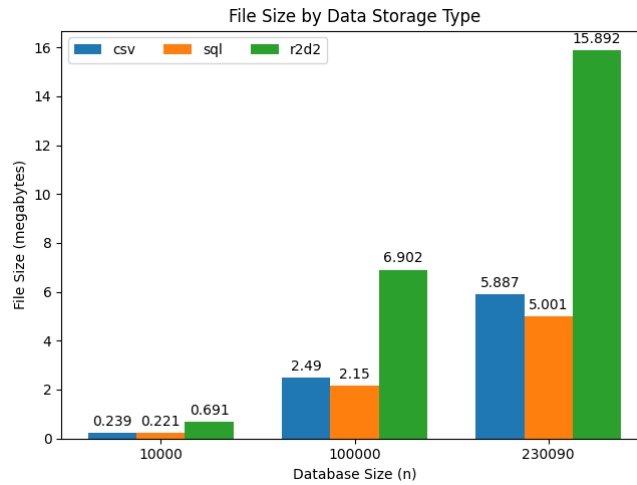


Figure 10: File sizes in **megabytes** for $n=10000$, $n=100000$, and $n=230090$. $n=10000$ is shared with Figure 9 for visual reference.