Scaling Stack Trace Fingerprinting

Mounir El Kirafi
mkirafi@os3.nl
University of Amsterdam

Supervisor: Luc Gommans (X41 D-Sec)

July 5, 2021

Abstract

A stack trace, the active stack frames at the point of crash, can contain much useful debugging information. However, this same information can be useful when penetration testing. More specifically, it can point to the usage of a specific framework and version, which in turn can lead to its vulnerabilities. Consequently, a tool to fingerprint such stack traces is useful. There currently exists such a tool for Java (X41 BeanStack), but due to the common use of JavaScript in web applications, there is a desire to extend this to JavaScript. The tool works by extracting relevant data from Java class files and inserting that data into a database. Data extracted from stack traces can then be searched for in that database. However, there is a scalability issue. With there already being 49 million rows in the database and each stack trace containing multiple stack frames, the querying time can quickly grow. That is why this research focuses on improving the current X41 BeanStack database have a more efficient storage and querying system, as well as to work with JavaScript. To do so, the parsing of stack traces and extraction of data from frameworks is first focused on. Then the research takes the current implementation and new use case to evaluate against possible database and storage methods. Finally, further improvements such as a better data model are also considered. Using the same database, MariaDB, but with a better data model, resulted in a speedup of 2-5.5x depending on the number of sequential queries. Further performance could be gained by switching to a hash index in PostgreSQL and when combining multiple query searches into a single query. Finally, it was also observed that using a NoSQL key-value database such as Couchbase improved performance even further.

1 Introduction

When developing (web) applications, errors often occur due to programming mistakes. To debug these, developers can use stack traces, which are the stack frames active at the point of the crash [6]. They contain the list of nested functions used, allowing a developer to follow that chain to the point of origin and find the bug. This has uses in end products, where a user can report a stack trace to the developer such that they can more easily fix the issue. However, from a security perspective a stack trace can leak information [19]. This is why they are often not shown. As traces can show the chain of functions used and in what files, this could lead to figuring out what libraries are used, and more specifically what version. This can then be linked with known vulnerabilities for that specific library version to perform an attack.
Fingerprinting techniques are a useful tool for a penetration tester, because they can reveal information to the attacker that can be exploited. For example, scanners such as Nmap [12] can guess the underlying OS used through various factors, or specific application through identifiers in the network traffic. This is also why a tool for fingerprinting used libraries through stack traces can be very useful. The main requirement for such a tool is the underlying database in which the attacker can cross-reference the function calls with those contained in libraries, along with the versions and CVEs [7] of those libraries.
There currently exists such a tool for Java fingerprinting through stack traces: X41 BeanStack [21]. It can process a Java stack trace and output the matched products, versions and their corresponding CVE(s). It should be noted that functions don't have to change between versions so it can be hard to match to a specific version. It works by first populating the database with function call locations extracted from Java classes. To match a stack trace, each line is analysed for the mentioned function and searched in the database. The result gets displayed as earlier mentioned.
This works well when the number of libraries is small, but as soon as that number starts to grow scaling issues start to occur. This issue gets amplified due to the fact that every single line in the stack trace needs to be searched (to find all libraries used), which means that the search time can get much larger very quickly. There is an interest to extend this tool to be used for JavaScript, due to the common use of JavaScript in web applications. This is what this research aims to accomplish. However, it has a considerable number of libraries, putting a large emphasis on the scalability issue. This is where new search and storage solutions can greatly benefit the tool. In this research, storage solutions and techniques such as SQL and NoSQL as well as optimizations such as hashing and indexing are evaluated. More specifically, the benchmark is the speed at which a stack trace can be searched in the database, such that it is faster than the current implementation.

2 Research Questions

Because the focus is the improvement of the current system with regards to the previously mentioned benchmark, the main research question of this paper is:

> ***How can the current X41 BeanStack stack trace fingerprinting database be improved for a more efficient storage and querying system?***

This paper focuses on answering the main question by addressing the following set of sub-questions:

- What is the necessary information from JavaScript libraries to populate the database and how can this be extracted?

- What are more optimal database storage solutions for the storage and querying of stack traces?
- How can these solutions be adjusted for better performance in the target use case?

2.1 Structure

To answer these questions, this thesis adheres to the following structure. Section *Related Work* reviews existing Stack Trace Fingerprinting (STF) implementations as well as bringing general database performance research into context. Section *Parsing and Data Extraction* provides information on JavaScript stack traces as well as a framework data extraction technique with the aim to answer the first sub question. Then section *Databases* will provide support in answering the second and third sub questions. This is first done in subsection *Current Beanstack Implementation* where the current implementation is explained. Then in *Database Types* various database types are evaluated and a choice is made based on the use case. Finally, in subsection *Further Improvements* further improvements are made based on an improved data model. Moving on to section *Method*, the testing methodology is explained. Then in section *Results* the results for these tests are shown. Finally, in sections *Discussion* and *Conclusion* these results are discussed and used to conclude an answer to the main research question.

3 Related Work

As mentioned in the introduction, there exists a Java version of the fingerprinting database, made by X41 D-Sec, called X41 BeanStack [21]. This is talked about in a presentation in 2013 by one of the developers, Eric Sesterhenn [1]. The tool is the concept on which the research is based. The current implementation uses indexing for a speedup in querying. It also combines data into several tables to reduce `JOIN` operations. However, this results in more duplicate data. The stack traces can be hashed prior to submission, although for anonymity rather than a faster lookup.
Another Java STF implementation was created by Stefan Schlott, as presented in the 2014 Easterhegg talk [15][16]. The implementation seemed to have the same concept as X41 BeanStack: extracting the functions from the Java classes and storing them in a database for lookup. However, looking at the source code and the talk, it did not appear to do any optimizations with regards to searching. It uses the Slick database querying and access library for Scala (Functional Relational Mapping) with an underlying unspecified SQL database.
Besides these two pieces of work, no other relevant STF work could be found. However, it should be noted that this research can also benefit from more general fast storage methods.
Research done by M. Indrawan-Santiago [9] indicates that NoSQL databases can deliver a performance benefit over more traditional SQL databases without the need for query optimization. This is due to the design of a NoSQL database being tailored to the specific data processing needs of the user. Further research done into the differences between SQL and NoSQL databases [11] indicates that some NoSQL databases are optimized for key-value stores, although not all perform as well, or even better than an SQL database. Although, Couchbase and MongoDB, two NoSQL key-value databases, do perform faster for read, write and deletion operations. This shows the need to properly pick a database for the specific use case. Other research also shows Couchbase as consistently performing well in terms of reading, when compared to other NoSQL databases [18].

4 Parsing And Data Extraction

To be able to use an STF tool, there must be a way to extract the required lookup data from stack traces, as well as a way to populate the database with information extracted from existing frameworks.

4.1 JavaScript Stack Traces

The input data for the lookup tool is always a stack trace. This is something dependable. Though, what can change is the format of the stack trace depending on the JavaScript engine used. These engines are the programs used to execute the JavaScript code and are generally developed by the major browser vendors. For example, all Chromium-based browsers (such as Chrome and Microsoft Edge) use V8 [20], Firefox uses SpiderMonkey [17], Safari uses JavaScriptCore [10] and Internet Explorer uses Chakra [2]. However, the focus lies more on back end uses of JavaScript. This is why the main syntax that is looked at is that of Node.js. "Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine" as stated on the Node.js website [13]. The format of a Node.js stack trace follows a set format. The first line contains the error class name, followed by the error message: `<error class name>: <error message>` [14]. What follows next is a series of stack frames, each containing a description of the location where the error occurred.

```
Error: Things keep happening!
   at /home/gbusey/file.js:525:2
   at Frobnicator.refrobulate (/home/gbusey/business-logic.js:424:21)
   at Actor.<anonymous> (/home/gbusey/actors.js:400:8)
   at increaseSynergy (/home/gbusey/actors.js:701:6)
```

Listing 1: Node.js stack trace example

In listing 1 an example stack trace is given. For each stack frame, the function name as well as the location in the corresponding file is given by indicating the filename, line number and column number. But, as seen in the first stack frame, sometimes no suitable function name can be found. This means that the only information remaining is the file and the location within the file. Because of this, when trying to fingerprint a stack trace, it is sometimes required to execute a fuzzy search without the function name.
To extract the data from the stack trace, regex can be used. Each stack trace line can be passed through the regex rule specified in listing 4.1:

```
/\sat ([a-zA-Z.0-9_]+ )?(\[as [a-zA-Z0-9_]+\] ↪
    )?\(?[a-zA-Z0-9\/_.-]+:[1-9][0-9]*:[0-9]+\)?/
```

- `\sat` - match the "at"
- `([a-zA-Z.0-9_]+ )?` - match the function name, might not exist
- `(\[as [a-zA-Z0-9_]+\] )?` - match the other possible function name
- `\(?[a-zA-Z0-9\/_.-]+:[1-9][0-9]*:[0-9]+\)?` - match the file path along with the file name, line number and column number

4.2 Database Population

Given the data extracted from a stack trace, one still needs to be able to look it up in a database. This database would consist of the data from a stack trace, as well as the information being looked for - the framework name, version number and CVEs. To be able to populate such as database, data needs to be extracted from JavaScript files. More specifically the different function names and their locations. One possibility would be to go through each JavaScript file and parse them manually. Though that is not needed, since a tool already exists to extract function calls from JavaScript files: *source maps*. These source maps are generally used to convert transformed/minified JavaScript code to the original source code. To do this, they must contain the function names, variable names and their original locations. This is exactly what is needed for the fingerprinting database. By generating a source map for a JavaScript file, it is essentially parsed and then the relevant data can be extracted. In listing 2 an example can be seen of how to generate and use such a source map.

```
 1  // Read file
 2  var data = fs.readFileSync(file, {encoding: "utf-8"});
 3  // Generate source map
 4  map = UglifyJS.minify(files, {sourceMap: {}}).map;
 5  // Get sourcemap consumer
 6  sourcemap.SourceMapConsumer.with(map, null, consumer => {
 7      // Loop over each found call
 8      consumer.eachMapping(function(call) {
 9      // Retrieve and log each relevant item
10      console.log("file:", call.source, "\t", "line:", call.originalLine, "\t", ↪
            "column:", call.originalColumn, "\t", "name:", call.name);
11    });
12  });
```

Listing 2: Generating a source map from a file

To actually fill the database with the extracted calls, the data logged in line 10 would actually be inserted into the database.

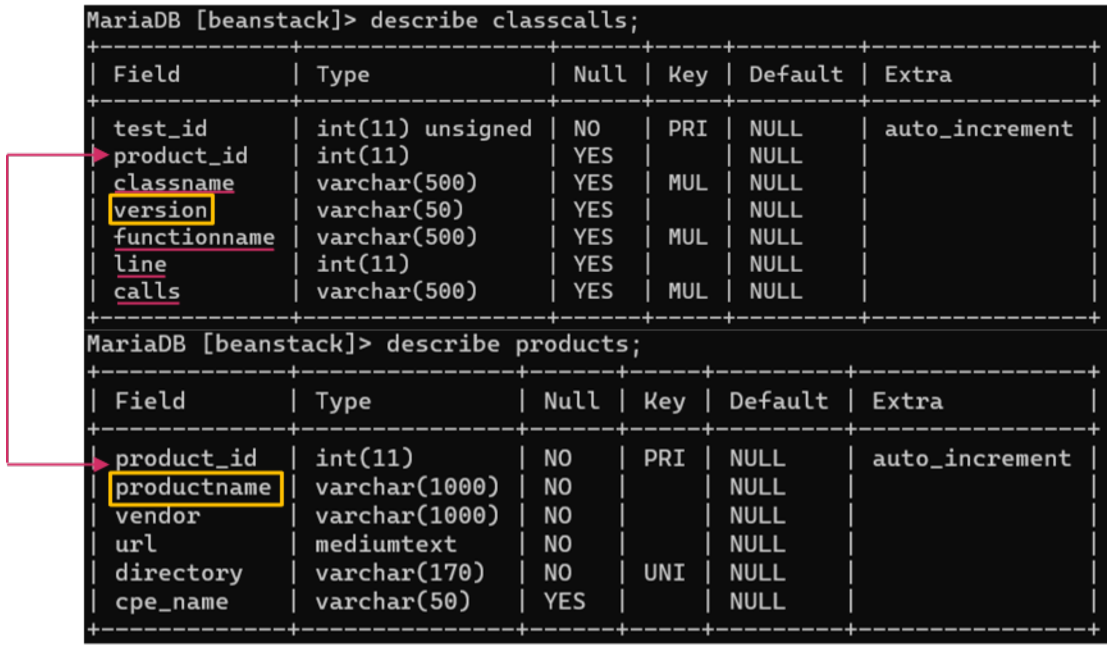
5 Databases

As indicated in the previous section, the necessary information to search for is the file name, line number, column number and the function name if available. With this we have a specific use case to which we can tailor a database choice and further optimizations. However, to do so, the current Java BeanStack implementation is reviewed first. Also, different databases and storage types are evaluated based on their features and design types.

5.1 Current BeanStack Implementation

The current X41 BeanStack [21] implementation works by allowing a user to input a stack trace through an API. When a fingerprinting request is sent, this stack trace is sent to the back end server which processes it. It is parsed for the relevant information and then searched in the database. Then the relevant name, its associated version numbers and the CVEs corresponding to the combinations of those two are returned to the user. It should be noted that the system can return multiple version numbers per framework since a function does not necessarily change between versions. This also results in multiple potential CVEs.
Looking deeper at the back end, the database in which the data is searched is MariaDB. This is a

relational SQL-based database. The data model consists of multiple tables, each containing information about a section of the data. The two that are most relevant however, are the `classcalls` and `products` table:



```
MariaDB [beanstack]> describe classcalls;
+--------------+------------------+------+-----+---------+----------------+
| Field        | Type             | Null | Key | Default | Extra          |
+--------------+------------------+------+-----+---------+----------------+
| test_id      | int(11) unsigned | NO   | PRI | NULL    | auto_increment |
| product_id   | int(11)          | YES  |     | NULL    |                |
| classname    | varchar(500)     | YES  | MUL | NULL    |                |
| version      | varchar(50)      | YES  |     | NULL    |                |
| functionname | varchar(500)     | YES  | MUL | NULL    |                |
| line         | int(11)          | YES  |     | NULL    |                |
| calls        | varchar(500)     | YES  | MUL | NULL    |                |
+--------------+------------------+------+-----+---------+----------------+
MariaDB [beanstack]> describe products;
+-------------+---------------+------+-----+---------+----------------+
| Field       | Type          | Null | Key | Default | Extra          |
+-------------+---------------+------+-----+---------+----------------+
| product_id  | int(11)       | NO   | PRI | NULL    | auto_increment |
| productname | varchar(1000) | NO   |     | NULL    |                |
| vendor      | varchar(1000) | NO   |     | NULL    |                |
| url         | mediumtext    | NO   |     | NULL    |                |
| directory   | varchar(170)  | NO   | UNI | NULL    |                |
| cpe_name    | varchar(50)   | YES  |     | NULL    |                |
+-------------+---------------+------+-----+---------+----------------+
```

Figure 1: `Classcalls` and `products` tables of the BeanStack MariaDB database

The values underlined in red are the ones that would be searched for. Then the values in the yellow box are the values the user gets back. These tables are linked using the `product_id`. As we can see here, we have two different tables and multiple columns to search through, still excluding the separate search for the CVEs. To speed up searches, the database uses indexes on the `classcalls` table. Specifically, on the `test_id`, `classname`, `functionname` and `class` columns, using a `B-Tree` index. Indices will be elaborated on in the next sub-section, but for now it is enough to know that they change the storage structure to be more efficient. A non-indexed lookup in an SQL database (based on the storage engine) does a full sequential search through all records as stored in disk/memory.

5.2 Database Types

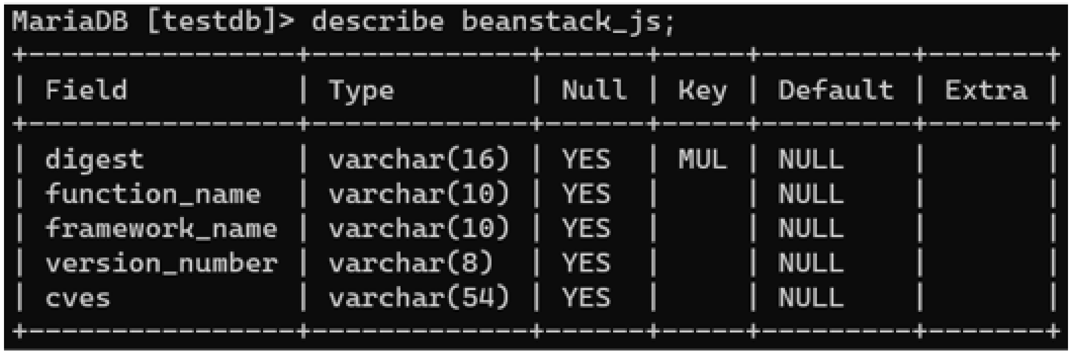
There are different database types, although the ones that concern this research the most are relational databases and non-relational databases, generally based on SQL and NoSQL respectively.
Relational databases, like the one used in the BeanStack back end, allow for highly structured data with relationships. This data is organized using a set of tables, with each table containing rows and columns. A relationship between the data in the tables can then be established through primary and foreign keys. By using these constraints as well as others such as requiring fields to not be 'NULL', a user can guarantee data integrity. Because of the relational nature and the data integrity,

relational databases are a proper solution for complex queries and data analysis. Examples of a relational database are Oracle, PostgreSQL, MySQL and MariaDB (fork of MySQL).
The specific way data is stored, accessed and managed is determined by the storage engine used. These engines are used for the CRUD (Create, Read, Update and Delete) operations of a database. For example, the two most used ones for MySQL are MyISAM and InnoDB. The main differences between these engines are generally different supported features such as foreign keys. Also, these engines can have different implementations of the underlying data structure which can impact performance. The most relevant difference for this research is the supported types of indices.
As mentioned in the previous subsection, the current MariaDB implementation uses a `B-tree` index. This is using the InnoDB engine. What this means is that the engine creates a new data structure called a B-tree to store the data in. In a `B-tree` (a self-balancing tree), data is kept in a sorted order. By combining the tree structure and the sorted order, data can be searched for with an average time complexity of $O(logN)$. This means that if there are N items to look through, on average, it will take $logN$ items to search through before finding the correct item. For a 'normal' search, this would result in an average time complexity of $O(N)$, since all records must be searched through sequentially.
A `B-tree` is excellent for most database use cases, especially when working with ranges rather than specific values. However, in the current use case the exact items are known, and no ranges are required. Because of this a hash index is more optimal. In theory this has an average searching time complexity of $O(1)$, meaning constant lookup time, regardless of database size. This is because the known key can be computed to a unique hash and point to the specific area where the data is stored. But this is I/O limited.
The problem with hash indices is that most SQL-based databases don't have a storage engine that supports it. For example, the only one that does in MySQL/MariaDB is the `MEMORY` storage engine. The problem with this is that it stores all the data in memory. In terms of performance this is very good but as the number of data increases, this is not practically scalable. Because of this limitation PostgreSQL was chosen. It allows for a hash-based index to be used without keeping all data in memory by using a different storage engine.

NoSQL on the other hand, which stands for Not only SQL, is a collection of database types that differ from relational databases. The meaning of the name in fact refers to the idea that relational databases are not suited for every single job. A NoSQL database generally has more flexibility in terms of data types per data structure. Each different NoSQL database type handles this differently.
For example, the key-value database is a type of database which, as the name suggests, stores values based on a key. The underlying data structure is essentially a hash table [5]. The value accessed by the key has fields, each of which containing their own data. These fields correspond to columns in a relational database. This is essentially the same as using a hash index, except that the whole database is created to follow this model. There are also other types of NoSQL databases (such as wide column, document, graph and object oriented), However, these are not relevant to the current use case. Based on research presented in the related work section (section 3) and performance research by Y. Li and S. Manoharan[11], Couchbase was chosen. In the related research, occasionally there were other databases that performed better, such as Redis or Cassandra. Although, Redis is an in-memory database which is excluded for the same reasons as the in-memory hash index storage engines. Cassandra is a wide column store database which does not fit the use case as well as a key-value store.

5.3 Further improvements

A key-value store or a hash index generally works best with a single key due to less searching complexity. However, in the current use case there is the file name, column number, line number and function name. To be able to search for all of those at once, they need to be combined into a single value. The most obvious solution would be concatenation with a separator. A better solution is generating a hash out of the concatenated strings. With this the resulting string is shorter as well as standardized in terms of length. To still allow for fuzzy searches where the function name is not known, the function name is added to the hash. Since it is not known whether a fuzzy search needs to be executed at the time of hashing, it is stored as a separate column to filter for after querying. For the hashing, the non-cryptographic hashing algorithm xxHash is used [3]. It features high speed hashing [8] with very low collision rates [4]. With the function used for this research (`xxhash.xxh64_hexdigest(<string to hash>)`), it produces a 16-character hexadecimal hash value.
Other than facilitating hash-based indexing and the use of key-value stores, using a single lookup value can also benefit the `B-tree` indexed data model. By replacing all the search columns with a single one, query complexity and searched data is decreased. This can be taken a step further by combining the different tables into a single one. With these changes, the multiple column search is reduced into a single one, and there is no more need to link results through multiple tables. This can be illustrated in the following new data model:



```
MariaDB [testdb]> describe beanstack_js;
+-----------------+-------------+------+-----+---------+-------+
| Field           | Type        | Null | Key | Default | Extra |
+-----------------+-------------+------+-----+---------+-------+
| digest          | varchar(16) | YES  | MUL | NULL    |       |
| function_name   | varchar(10) | YES  |     | NULL    |       |
| framework_name  | varchar(10) | YES  |     | NULL    |       |
| version_number  | varchar(8)  | YES  |     | NULL    |       |
| cves            | varchar(54) | YES  |     | NULL    |       |
+-----------------+-------------+------+-----+---------+-------+
```

Figure 2: New data model with columns combined into single hash value and tables combined

Another aspect to consider is the query used to retrieve the data. With the new data model, such a query would look like:

```
select * from beanstack_js where digest='testdigest'
```

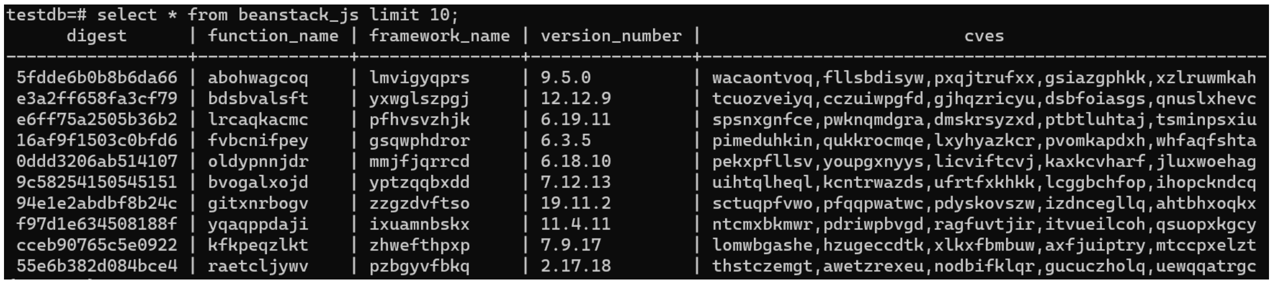
This query would then run sequentially for every parsed stack trace line. Let this be 2 runs for this example. If this query would be altered to combine the lookups in a single query, this could potentially shift the load more from individual query processing, to database lookup speed and effectively increase I/O utilization. Such a query would then look like:

```
select * from beanstack_js where digest='testdigest' or digest='testdigest2'
```

By scaling this to even more searched keys in a single query, a speedup could potentially be observed.

6 Method

Given all the aspects that can result in a speedup, this research focuses on testing four main potential improvements. These improvements aim to answer the last two research sub questions. It should be noted that for all tests and databases, a self generated database of random strings was used. This is due to the lack of an existing real database with an adequate size. The strings were made such that they would be relatively representative in terms of character count. An example can be seen in figure 3:



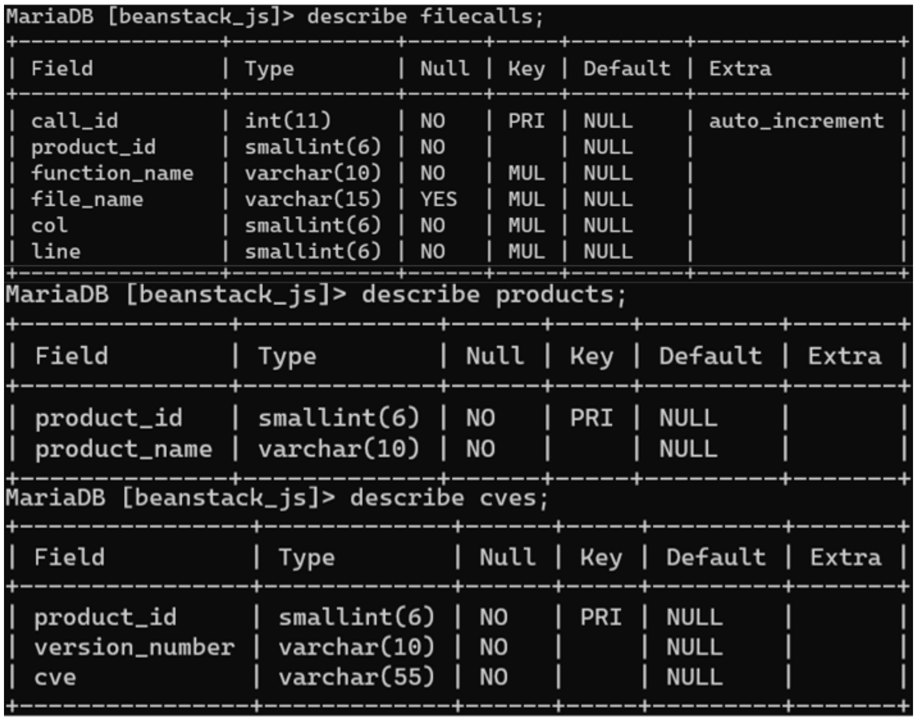
```
testdb=# select * from beanstack_js limit 10;
      digest      | function_name | framework_name | version_number |                          cves
------------------+---------------+----------------+----------------+-------------------------------------------------------
 5fdde6b0b8b6da66 | abohwagcoq    | lmvigyqprs     | 9.5.0          | wacaontvoq,fllsbdisyw,pxqjtrufxx,gsiazgphkk,xzlruwmkah
 e3a2ff658fa3cf79 | bdsbvalsft    | yxwglszpgj     | 12.12.9        | tcuozveiyq,cczuiwpgfd,gjhqzricyu,dsbfoiasgs,qnuslxhevc
 e6ff75a2505b36b2 | lrcaqkacmc    | pfhvsvzhjk     | 6.19.11        | spsnxgnfce,pwknqmdgra,dmskrsyzxd,ptbtluhtaj,tsminpsxiu
 16af9f1503c0bfd6 | fvbcnifpey    | gsqwphdror     | 6.3.5          | pimeduhkin,qukkrocmqe,lxyhyazkcr,pvomkapdxh,whfaqfshta
 0ddd3206ab514107 | oldypnnjdr    | mmjfjqrrcd     | 6.18.10        | pekxpfllsv,youpgxnyys,licviftcvj,kaxkcvharf,jluxwoehag
 9c58254150545151 | bvogalxojd    | yptzqqbxdd     | 7.12.13        | uihtqlheql,kcntrwazds,ufrtfxkhkk,lcggbchfop,ihopckndcq
 94e1e2abdbf8b24c | gitxnrbogv    | zzgzdvftso     | 19.11.2        | sctuqpfvwo,pfqqpwatwc,pdyskovszw,izdncegllq,ahtbhxoqkx
 f97d1e634508188f | yqaqppdaji    | ixuamnbskx     | 11.4.11        | ntcmxbkmwr,pdriwpbvgd,ragfuvtjir,itvueilcoh,qsuopxkgcy
 cceb90765c5e0922 | kfkpeqzlkt    | zhwefthpxp     | 7.9.17         | lomwbgashe,hzugeccdtk,xlkxfbmbuw,axfjuiptry,mtccpxelzt
 55e6b382d084bce4 | raetcljywv    | pzbgyvfbkq     | 2.17.18        | thstczemgt,awetzrexeu,nodbifklqr,gucuczholq,uewqqatrgc
```

Figure 3: Example data from self-generated database

For the creation, the following values were used:

- 15-character file name with random letters between a-z
- random column number between 0 and 120
- random line number between 0 and 10000
- 10-character function name with random letters between a-z
- 10-character framework name with random letters between a-z
- version number with three randomized values between 0 and 20 separated by a dot
- 5 random 10-character CVE names with random letters between a-z

The file name, column number and line number were concatenated with the separator '-' before being hashed using xxHash[3]. This was repeated 100 million times for as many unique (in terms of combination of relevant search data) data records. For the new data model, a hash digest of the file name, column number and line number were taken with the xxHash algorithm [3].
For the first potential improvement, the difference between the data models is tested. For a fair comparison, both tests will be run on MariaDB with the same setup. To test the difference a data model was created for JavaScript to reflect the "old" data model in the Java implementation:



```
MariaDB [beanstack_js]> describe filecalls;
+---------------+-------------+------+-----+---------+----------------+
| Field         | Type        | Null | Key | Default | Extra          |
+---------------+-------------+------+-----+---------+----------------+
| call_id       | int(11)     | NO   | PRI | NULL    | auto_increment |
| product_id    | smallint(6) | NO   |     | NULL    |                |
| function_name | varchar(10) | NO   | MUL | NULL    |                |
| file_name     | varchar(15) | YES  | MUL | NULL    |                |
| col           | smallint(6) | NO   | MUL | NULL    |                |
| line          | smallint(6) | NO   | MUL | NULL    |                |
+---------------+-------------+------+-----+---------+----------------+
MariaDB [beanstack_js]> describe products;
+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
| product_id   | smallint(6) | NO   | PRI | NULL    |       |
| product_name | varchar(10) | NO   |     | NULL    |       |
+--------------+-------------+------+-----+---------+-------+
MariaDB [beanstack_js]> describe cves;
+----------------+-------------+------+-----+---------+-------+
| Field          | Type        | Null | Key | Default | Extra |
+----------------+-------------+------+-----+---------+-------+
| product_id     | smallint(6) | NO   | PRI | NULL    |       |
| version_number | varchar(10) | NO   |     | NULL    |       |
| cve            | varchar(55) | NO   |     | NULL    |       |
+----------------+-------------+------+-----+---------+-------+
```

Figure 4: Old data model recreated for JavaScript implementation

This model has three tables, one containing the stack trace information, one containing the framework information and one containing the corresponding CVEs. The tables are all linked with the `product_id` value. It was created this way to represent the need to search through multiple tables and columns at the same time. Then, as seen in figure 2 and explained in section 5.3, the new data model was also applied.
The second test aims to find out how much of a speedup querying multiple keys at once versus multiple queries with a single key delivers. This is done on PostgreSQL
The third test is to evaluate the speedup of using a hash index along with the new model. PostgreSQL is used for this. Because of results that will become apparent in the next section, this test is done with 20 digests per single query.
The final test evaluates the speedup that a NoSQL key-value database provides. As such Couchbase is used to perform lookup tests. These tests will first look at how different lookup methods within Couchbase affect performance. This is because Couchbase offers the use of a query, as well as the use of its own key-value API for direct data retrieval. Then finally its performance is compared to MariaDB's best performance with the new model and multiple keys per query. PostgreSQL is also included in this comparison, with its best performance with the new model, hash indexing and multiple keys per query.
For all tests, to facilitate a fair comparison, caching was avoided as much as possible. This is because the database is sufficiently large that it can be expected that a relatively large number of values will not be in memory. For MariaDB and PostgreSQL the cache was both flushed prior to each run. But for Couchbase, this was not possible due to how integral caching is in the design.

This is why for each run data from a different and non-overlapping area was queried. There would be a difference of 10-20 million records to ensure that the values requested would not be from the cache. Caches based on recently used values are also avoided since every searched value is unique. Nevertheless, even with these measures, during testing a cache miss ratio of 95%+ could be observed, meaning that some values were still being retrieved from cache. Although this should be kept in mind when viewing the results, it does not impact the results significantly enough to make them unreliable.
Furthermore, for each test different amounts of total queries are tested. The general format will be 10 queries, then 100 queries, then 1000 queries, then 10,000 queries and so on as needed. The exact amounts can be seen in the resulting graphs. Each test with a specific number of total queries is run 10 times, and the average of those times is taken and displayed.
Finally, all tests and databases ran on a 4-core Xeon E-2124 at 4.3GHz, using 16GB of memory and running on Ubuntu 20.04.2 LTS x86_64. All data was stored on a 500GB Samsung 860 EVO SATA SSD.

7 Results

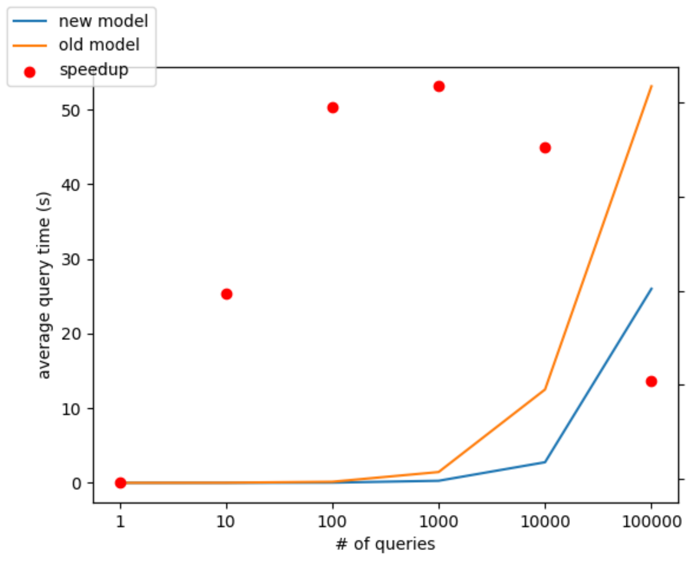


Figure 5: Average query time of the new and old data model with multiple numbers of sequential queries, as well as relative speedup of $\frac{old}{new}$ model

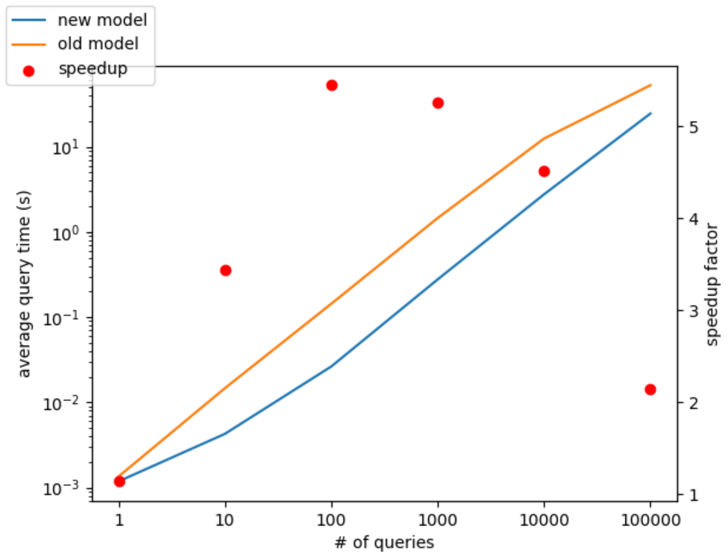


Figure 6: Figure 5 except with logarithmic scaling

In figures 5 and 6 the results of the performance tests of the different data models for MariaDB can be see. On the left axis the average query time is given, while the right axis denotes the speedup factor (red dots). This is calculated by dividing the average query time of the old model by the average query time of the new model, for each query quantity. The left figure shows the absolute timings for both models, while the right figure shows the timings on a logarithmic scale. This allows for the relative relationship to be seen better. In figure 6 it can be seen that both models grow linearly, which means that both timings consistently grow by around 10x for every time the query quantity grows by 10x. However, this starts to drop at the 100,000 queries amount for the old model. This results in the speedup factor drastically dropping down to almost 2x, after peaking around 5.5x. The average query time of the new model stays consistently lower than that of the old model.

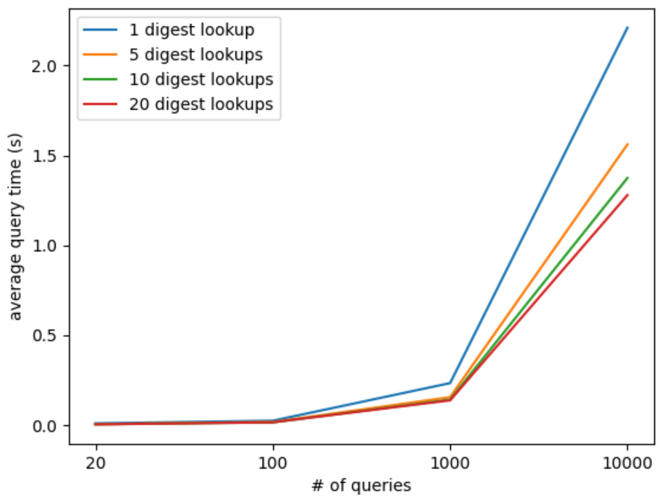


Figure 7: Average query time of different numbers of digests looked up in a single query with different total sequential queries executed

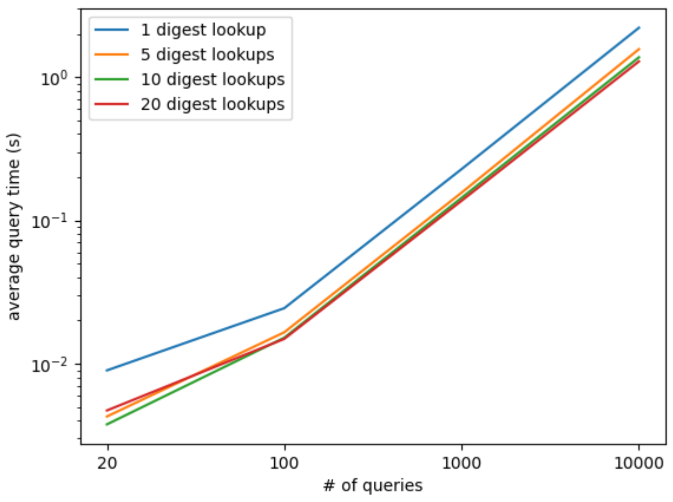


Figure 8: Average query time in log scale of different numbers of digests looked up in a single query with different total sequential queries executed

Figures 7 and 8 show the results of combining multiple digest lookups in a single query. This test features 1, 5, 10 and 20 digests per query. Because of this the minimum number of sequential queries is also raised to 20 as opposed to figures 5 and 6 because of the 20 digests in a single query. From the logarithmic scaled figure on the right linear scaling can once again be seen. In both figures an advantage in query time can be observed every time the number of digests per query is increased. For 10,000 sequential queries, the single digest per query takes 2.2 seconds. The 20 digests per query takes 1.3 seconds. However, both of these are still faster than MariaDB with the new model, which took 2.7 seconds for 10,000 sequential queries without any consolidation of queries. Due to the assumption that the performance gain is based on better I/O utilization, the disk read speed during the test was also measured, as seen in table 1:

Digests	Average Disk Read Speed (MB/s)
1	35
5	56
10	67
20	75

Table 1: Average disk read speed during testing of multiple digests per query

There is a clear increase in disk read speed as the digest number increases. The speed increases 2.14x from 1 to 20 digests.

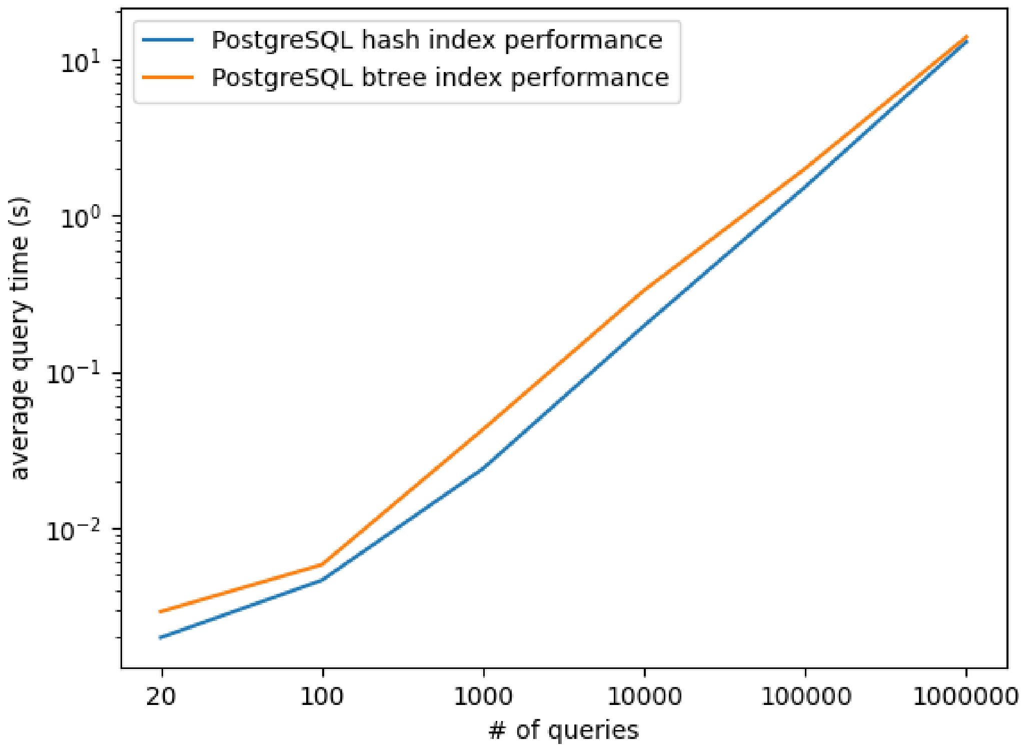


Figure 9: Performance comparison of hash index vs B-tree index in PostgreSQL

Digests	Hash index average query time	B-tree index average query time	speedup factor
20	0.001983	0.002907	1.466
100	0.004616	0.005807	1.258
1000	0.023896	0.042595	1.782
10000	0.196430	0.330711	1.684
100000	1.520870	1.989694	1.308
1000000	12.90908	13.83274	1.072

Table 2: Query timing (seconds) of different PostgreSQL indices along with speedup of hash over B-tree index

In figure 9 the performance of the hash and B-tree indices in PostgreSQL can be observed. The hash index is consistently faster, with a speedup of 1.25-1.78x up to 100,000 queries. However, at the 1,000,000 queries, this gap closes and the speedup factor drops to 1.072.

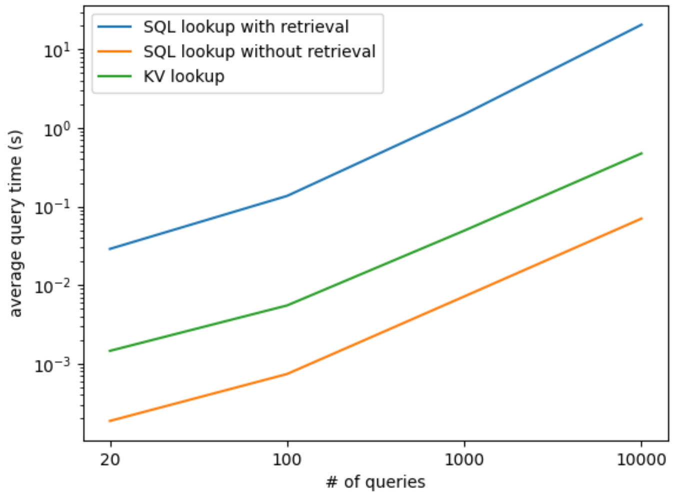


Figure 10: Comparing the different lookup methods for Couchbase

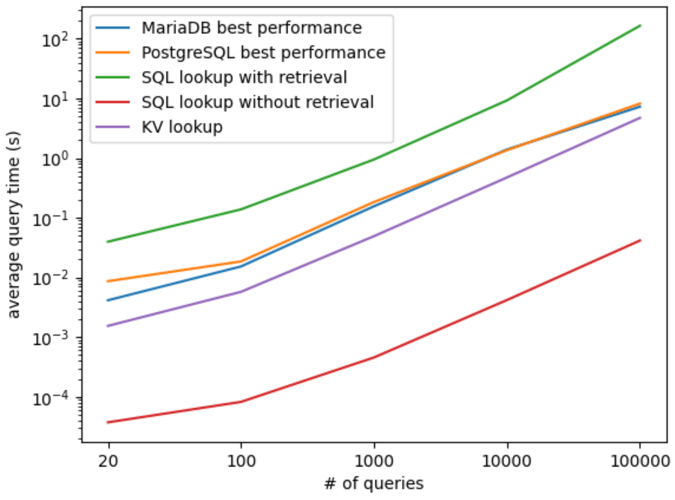


Figure 11: New data model MariaDB vs hash index, new data model and multiple digests per query for PostgreSQL vs Couchbase different lookup methods

The Couchbase Python client offers two different ways to interact with the database: using its SQL-like query language or using its key-value API. Figure 10 shows a performance comparison between these. However, when looking up data with the query a Python iterable is returned. In the figure it can be seen that simply requesting this is the fastest (SQL lookup without retrieval). But when trying to unpack this iterable to get the actual values, the total time increases significantly (two orders of magnitude, as can be seen by the log scale). The lookup through the key-value API sits in between these two results, being an order of magnitude slower than the query lookup without retrieval, but being an order of magnitude faster than the query lookup with retrieval of data.
In figure 11 all of the best cases per database are compared. First, PostgreSQL with a hash index, the new data model and 20 digests per query. Second, MariaDB is used with the new data model, as well as 20 digests per query. The benefits of this change also extend to MariaDB and Couchbase as can be seen in the appendix (figures 12 and 13). Finally, Couchbase with the different access methods. Couchbase's SQL-like lookup is the slowest. PostgreSQL and MariaDB are very close, with MariaDB being slightly faster. This can be attributed to MariaDB's B-tree implementation being faster than that of PostgreSQL. Combining the benefits of having multiple digests per query, it is slightly faster than also using a hash index in PostgreSQL. The next large jump belongs to the key-value API lookup of Couchbase. Finally, the query lookup without retrieving the data is the fastest. However, in terms of real-world usage, the key-value lookup of Couchbase is the fastest.

8 Discussion

During the research and testing, caching was avoided as much as possible. This was to ensure a fair comparison, and not rely on whether one database happened to have that piece of data in cache or not. However, the use of caching should still be acknowledged as a method of increasing lookup speed. Generally, two types of caches can be used here: most used data caching and recently used data caching. Both can contribute to the case where many users all share one or more frameworks used, resulting in many of the same lookups being done. Another similar suggestion would be to order the database by most used to least used data, to ensure that the most used data can be accessed faster. However, as can be seen in table 3 in the appendix, the lookup time does not change, regardless of what section of the database is accessed.
One of the issues encountered during testing was related to the SQL-like lookup of Couchbase. Using the Python client library, one can input a query and get a Python iterable in return. In Python, iterables are generally fast to unpack/iterate through. However, this one was very slow. Although it is still not clear why exactly this happens, a solution could be to use a different programming language to interface with the database.
In terms of interesting results, in figure 11 it can be seen that MariaDB actually performs slightly better than PostgreSQL (although the difference is so little that it is almost negligible in most cases). This could at first sight indicate that the hash index of PostgreSQL does not deliver a performance improvement with regards to the B-tree implementation. However, when looking at figure 9, it can be seen that there actually is an improvement. This suggests that the B-tree performance of PostgreSQL is worse than that of MariaDB, but when combining the benefit of using a hash index as well as multiple digests per query, this evens out with MariaDB's B-tree index with multiple digests per query.

9 Conclusion

With the research and the results, the main research question can now be answered. This starts by answering the first sub question. By understanding the information that is extracted from libraries, a solid basis for the use case is created. This leads into the other two sub questions, where the database performance is improved. First of all through the use of a more read optimized data model. This stems from the consideration of the use case and how the required data can be used to get there. With a single table and a single value to search through a speedup of 2-5x could be gained. This better data model - in terms of querying speed - can also facilitate a further speedup when combined with a new storage system, better suited for the use case. This can for example be PostgreSQL with a hash index or Couchbase with a key-value store. Finally, further improvement can be gained through for example better query design. By combining multiple queries into a single query, better I/O utilization is achieved. All these combined answer the question of how the current X41 BeanStack STF database can be improved for a more efficient storage and querying system.

9.1 Future Work

There are still multiple ways to improve the current research. First of all, one could evaluate the speedup gained from distributing the database over multiple systems versus the speedup parallelism could bring. A distributed approach could split the workload over multiple systems, while a parallel approach on the same system could show CPU performance related factors. As seen in the results for the multiple keys in a single query, I/O utilization played a role there. As such, the effect of different hardware types (HDD, SATA SSD, NVMe SSD) could be useful to research. Finally, it would be valuable to assess whether using a different programming language for the query lookup for Couchbase could lead to a significant speedup (due to the issues with the slow Python iterable).

9.2 Ethical considerations

During this research, no private data or systems were used. The only point at which any of the research could intersect with user data or any ethical issue is when processing a submitted stack trace. However, their data is not stored. However, it should be noted that as with all security related research, once published a malicious party could abuse such a lookup tool for unethical purposes.

References

[1] *BeanStack lightning talk.* URL: https://media.ccc.de/v/30C3_-_5562_-_en_-_saal_g_-_201312281245_-_lightning_talks_day_2_-_nickfarr#t=2298.

[2] *ChakraCore.* URL: https://github.com/chakra-core/ChakraCore.

[3] Yann Collet. *xxHash.* URL: https://github.com/Cyan4973/xxHash.

[4] *Collision ration comparison.* URL: https://github.com/Cyan4973/xxHash/wiki/Collision-ratio-comparison.

[5] Wikipedia contributors. *Hash Table.* URL: https://en.wikipedia.org/wiki/Hash_table.

[6] Wikipedia contributors. *Stack trace.* URL: https://en.wikipedia.org/wiki/Stack_trace.

[7] *CVE.* URL: https://cve.mitre.org/.

[8] Yue Du. *PyPi xxHash 2.0.2.* URL: https://pypi.org/project/xxhash/.

[9] Maria Indrawan-Santiago. "Database Research: Are We at a Crossroad? Reflection on NoSQL". In: *2012 15th International Conference on Network-Based Information Systems.* 2012, pp. 45–51. DOI: 10.1109/NBiS.2012.95.

[10] *JavaScriptCore Documentation.* URL: https://developer.apple.com/documentation/javascriptcore.

[11] Yishan Li and Sathiamoorthy Manoharan. "A performance comparison of SQL and NoSQL databases". In: *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM).* IEEE. 2013, pp. 15–19.

[12] *Nmap.* URL: https://nmap.org/.

[13] *Node.js.* URL: https://nodejs.org/en/.

[14] *Node.js v16.4.1 documentation.* URL: https://nodejs.org/api/errors.html#errors_error_stack.

[15] Stefan Schlott. *Easterhegg Java Stacktrace Fingerprinting.* URL: https://media.ccc.de/v/EH2014_-_5633_-_de_-_degerloch_-_201404201345_-_java_stacktrace_fingerprinting_-_skyr.

[16] Stefan Schlott. *Easterhegg Java Stacktrace Fingerprinting Github source.* URL: https://github.com/Skyr/tracefp.

[17] *SpiderMonkey.* URL: https://spidermonkey.dev/.

[18] Enqing Tang and Yushun Fan. "Performance Comparison between Five NoSQL Databases". In: *2016 7th International Conference on Cloud Computing and Big Data (CCBD).* 2016, pp. 105–109. DOI: 10.1109/CCBD.2016.030.

[19] *Testing for Stack Traces.* URL: https://owasp.org/www-project-web-security-testing-guide/v41/4-Web_Application_Security_Testing/08-Testing_for_Error_Handling/02-Testing_for_Stack_Traces.

[20] *V8.* URL: https://v8.dev/.

[21] *X41 BeanStack: Java Fingerprinting using Stack Traces.* URL: https://beanstack.io/.

Appendix

Offset	MariaDB	PostgreSQL	Couchbase
0	0.001876	0.001983	0.001171
100000	0.001739	0.001893	0.001149
10000000	0.001901	0.001956	0.001297
50000000	0.001954	0.001981	0.001116

Table 3: Query time (seconds) to retrieve 20 values in different parts of each database

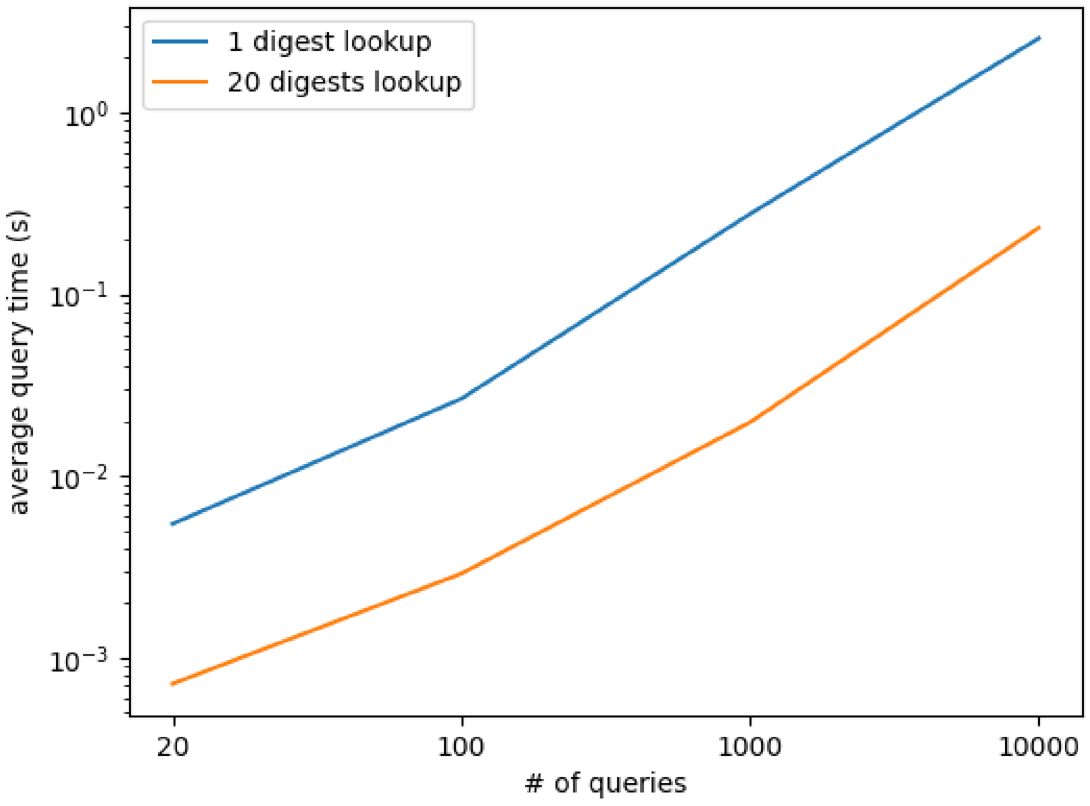


Figure 12: Using 1 and 20 digests per query for MariaDB

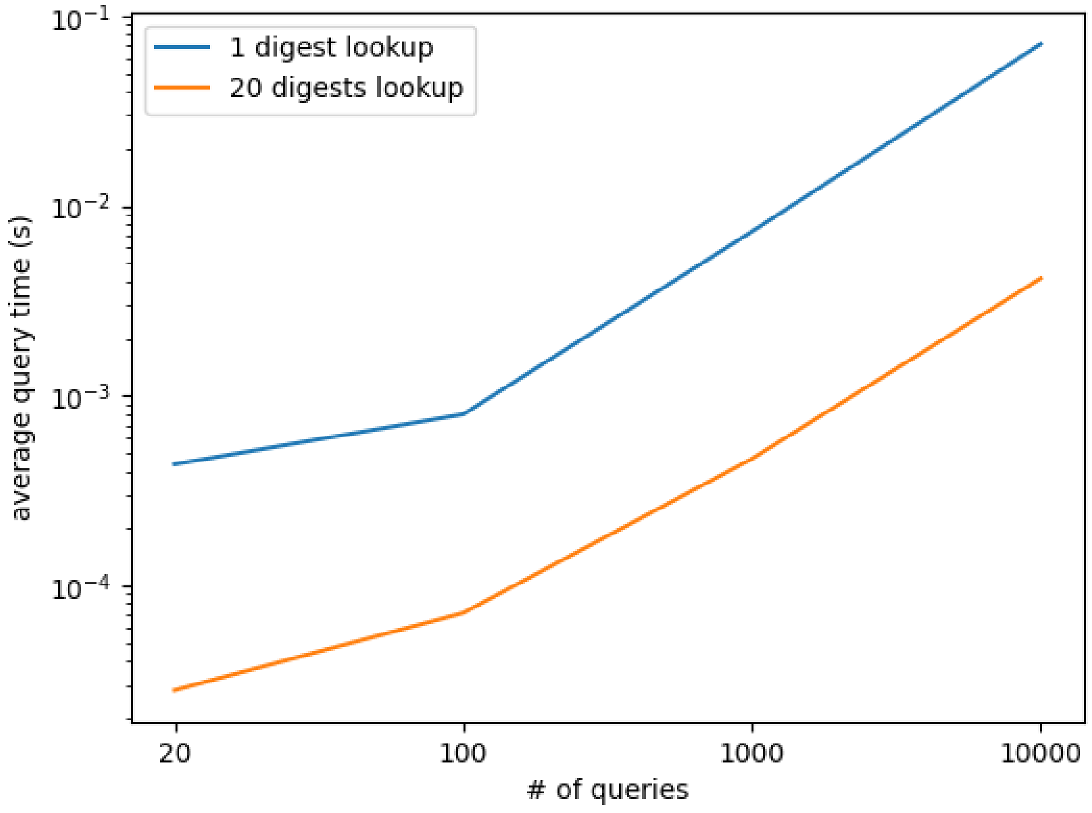


Figure 13: Using 1 and 20 digests per query for Couchbase