Department of Informatics, University of Zürich

BSc Thesis

Integrating the RCAS Index with the Software Heritage Archive

Marc Rettenbacher

Matrikelnummer: 17-727-843

Email: marc.rettenbacher@uzh.ch

July 15, 2020

supervised by Prof. Dr. M. Böhlen and K. Wellenzohn

University of Zurich[UZH]

Department of Informatics

Acknowledgements

I would like to express my gratitude to my supervisor Kevin Wellenzohn for always taking his time to provide valuable feedback and support throughout the whole process. I would also like to thank Prof. Dr. Michael Böhlen for the opportunity to write my thesis at the Database Technology Group at University of Zurich.

Abstract

The Software Heritage Archive aims to collect and preserve all publicly available software in the form of source code. It saves the project history and structure in the form of a graph and makes it publicly available through multiple interfaces. We want to use the data from the Software Heritage Archive to test the novel Robust Content-And-Structure (RCAS) index on a larger scale. This is a first step to providing an interface to query the Software Heritage Archive directly. For this we propose a way to parse the Archive and extract file paths and file sizes of all projects as sample values for both Content and Structure, as the current interfaces do not directly offer this functionality.

We implement and use a RCAS index to integrate our parsed data, as the index is made for semi-structured hierarchical data and thus answers CAS queries efficiently. To measure the index performance, we run different queries against it, utilizing the descendant axis // and the wildcard character $\star$ in the path part of the query. We found that the placement of the descendant axis // and wildcard $\star$ has a large impact on query performance.

Zusammenfassung

Das Ziel des Software Heritage Archives ist es, alle öffentlich verfügbare Software als Source Code zu sammeln und zu archivieren. Es speichert die Struktur und den Verlauf aller Projekte als Graph und macht diesen durch mehrere Schnittstellen öffentlich zugänglich. Wir wollen die Daten des Software Heritage Archives dazu nutzen, den neuartigen "Robust Content-And-Structure" Index mit einem grösseren Datenset zu testen. Dies ist der erste Schritt, um eine Schnittstelle für die Abfragung des Software Heritage Archives direkt zu erstellen. Wir stellen einen Ansatz dar, um Dateipfade und Dateigrössen als Beispiele für sowohl Struktur als auch Inhalt des Archives herauszulesen, da die jetzigen Schnittstellen diese Funktionalität nicht anbieten.

Wir implementieren und benutzen den RCAS Index um die analysierten Daten mit dem Index zu integrieren, da er für halbstrukturierte, hierarchische Daten gemacht wurde und daher CAS Abfragen effizient beantworten kann. Um die Performance des Indexes zu messen benutzen wir mehrere Abfragen, die unter anderem die "descendant axis $//$" und den "wildcard character $\star$" beinhalten. Wir lernen, dass die Position der "descendant axis $//$" und der "wildcard $\star$" in der Abfrage einen wichtigen Einfluss auf die Performance hat.

Contents

List of Figures

List of Tables

1. Introduction

With the growing trend of storing data digitally and often also publicly available, the access to rich and complex data becomes easier by the day. Examples being public posts on social media to be used in sentiment analysis, the world wide web itself or public code-repositories such as the Software Heritage Archive to analyse development trends.

In this thesis we use data from the Software Heritage Archive to test the RCAS index on a larger scale. According to the Software Heritage Website [6][7], the Archive itself is a project that aims to collect, preserve and share all publicly available software in the form of source code. It aims to help preserve our cultural heritage, support the research community and industry. The Software Heritage Archive archives a project by saving each commit individually, thus preserving the history of each project. It currently archives over 126 million different software projects, containing more than 8 billion source files and 1.7 billion commits. It provides access to the data through multiple ways:

- A web-application allows for easy access through the browser itself, allowing the user to search different repositories/source packages etc. and inspect those.
- An API for programmatic access to the content stored in a graph-structure. This allows the lookup of individual directories, commits etc.
- A downloadable database of the whole graph dataset and additional smaller subsets of it.

The above access methods are not sufficient for our needs, as we want to search for files using path-based queries that also restrict the values of an attribute, such as the file size. To overcome this, we use the downloadable database and propose a way to manually extract file paths and file sizes as outlined in Chapter 2. We then test our RCAS index implementation using selected Content-and-Structure queries.

We specifically use Content-and-Structure (CAS) queries as they are queries that pose restrictions on both the content and the structure of the dataset the query is executed on, both of which are important aspects of the Software Heritage Archive. For CAS queries we use `blue` for the structure and `red` for the content part of the query. In the context of the Software Heritage Archive we can imagine a query such as:

Q_1: All developers that modified a file inside the tests directory in the last ten days.

Here we restrict the content of the archive to only include commits with a timestamp that is less than 10 days old and we restrict the structure by requiring at least a test file

inside a tests directory to have changed in the commit. Or we could have a query Q_2: `//src/*, size<500` that simply searches for all files directly located in any `src` folder with a size smaller than 500 bytes. Here, the size restriction affects the content, and requiring the files to be directly located in any `src` folder affects the structure of the archive. We display a sample project that could be saved in the Software Heritage Archive in Table 1. We can then query all files with a size between `50k` and `100k` bytes in the `util` folder with a query Q_3: `/src/util//, 50k ≤size≤ 100k`, which returns the header file `/src/util/types.h` with size `66274` bytes. In Section 3.1 we define additional symbols, such as the descendant axis `//` and the wildcard character `*`.

File path	**File size (bytes)**
/.gitignore	122,624
/src/util/types.h	66,274
/src/util/helpers.h	135,595
/src/main.cpp	183,329
/src/merger.h	185,033
/src/merger.cpp	185,036

Table 1.: Sample files from a developer project in the Software Heritage Archive.

2. Software Heritage Archive

In this chapter we give an overview of the Software Heritage Archive structure and how the structure is reflected in the relational tables of the archive. We then discuss possible approaches to parse the archive database together with their respective advantages and disadvantages.

2.1. Software Heritage Structure

We summarize the Software Heritage Archive structure from [4] and [8]. The Software Heritage Archive is structured as a fully deduplicated Merkle Directed Acyclic Graph (Merkle DAG). A Merkle DAG uses cryptographically strong hashes as node identifiers. The identifiers of non-leaf nodes are based on the hashes of their child nodes [3]. The authors decided to use a DAG due to source code artefacts being massively duplicated across projects. In the DAG we can save each unique artefact only once and refer to it each time it is referenced in a different artefact. This allows them to track software artefacts across different projects and consequently reduces the storage size of the DAG. The DAG is saved as a set of relational tables, which can be organized in 5 logical layers (groups), as can be seen in Figure 1. A sixth group contains additional crawling information on when and where a snapshot was encountered. We describe the tables and their respective groups along with the number of rows of each table for the `popular-3k-python` dataset, which is a subset of the whole archive, as seen in Section 4.1. The top logical layer of the Archive consists of `snapshots`, which refer to each `revision` contained in them. Each `revision` points to its project directory structure, which points to the `content` tables of the `directory` files. We use **bold** for group names, `typewriter` for table names and *italic* for attribute names.

Snapshots are the top layer of the DAG, they capture the whole state of a project across all branches[1] for a given point in time. This also allows us to deduplicate unmodified forks in the archive, as we then have two different **snapshot** entries pointing to the same `revision`. Each `snapshot_branch` points to the latest revision of a branch.

- `snapshot_branches` (8.2M): An intermediate table to represent the many-to-many relation between `snapshot` and `snapshot_branch`.
- `snapshot`(10.3k): contains all snapshots captured by the archive.
- `snapshot_branch`(694k): contains all branches captured by the archive, the *target* attribute points to a given revision.

[1] Modern Version Control Systems, like git, allow developers to work on different versions, or *branches*, of the project in parallel.

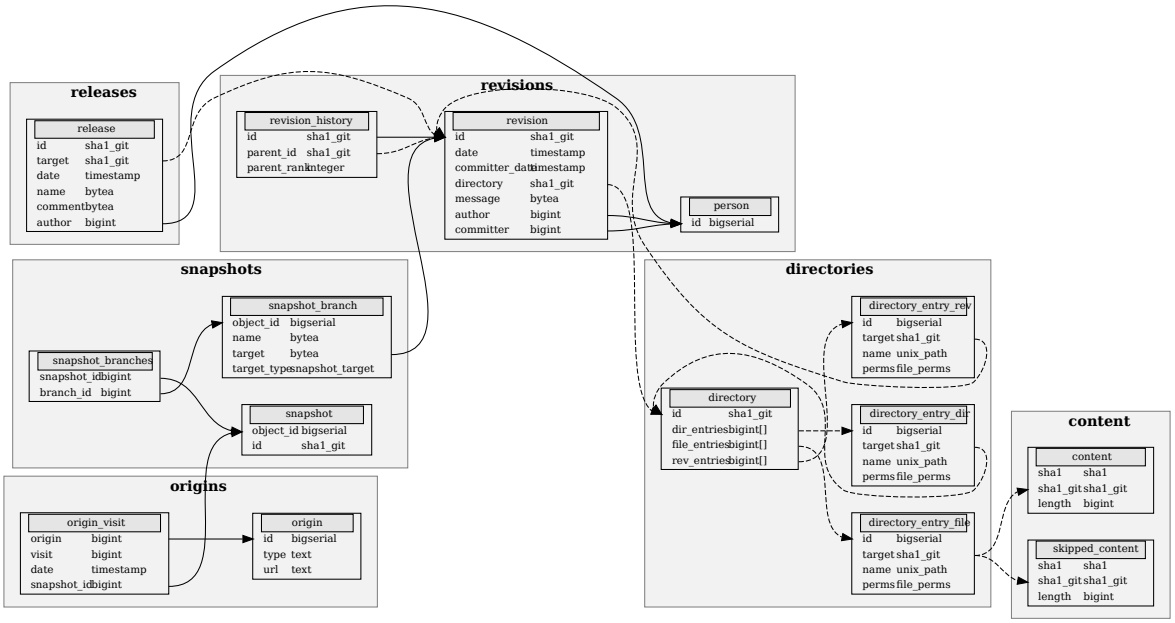

Figure 1: Software Heritage Relational Dataset, Image from [8].

Releases, i.e. tags, are tagged revisions, which mark important milestones such as a version release of the project etc. Each release points to its revision and may contain additional information such as date, name, comment, author.

- `release` (11.0k): Table referring to a tagged `revision` via *target* attribute and contains additional information about the release.

Revisions, also known as commits, refer to the state of the entire project source tree at a given point in time. Revisions cover only one branch, unlike snapshots. Each revision points to the root `directory` of its project.

- `revision_history` (5.8M): Contains the ordered set of previous revision ids (parents). The *parent_rank* attribute defines if a previous revision was an initial repository revision, a normal merge revision or any other type of revision.
- `revision` (5.2M): Contains all revisions stored in the archive along with revision information and the id of the root `directory`
- `person` (127k): Contains information about the revision author, but name and email were removed.

Directories contains all file paths by storing the individual directory and file names, as well as additional directory information. Each directory may point to its `content` if it contains any files. A directory may also include additional revision information, which is not relevant for our purposes.

- `directory` (17.0M): Each directory contained in the archive. It holds references to all of its subdirectories and also to all of its files, but no information about itself. The

file- and subdirectory-references are saved as an array of ids (see attributes *dir_entries* and *file_entries*).

- `directory_entry_dir` (12.8M): Contains the name and the permissions of the `directory` which the *target* attribute refers to.
- `directory_entry_file` (11.2M): Contains the name and the permissions of each file in the archive. Points to the **content** of the file via the *target* attribute.

Content describes the lowest layer of the DAG and forms its leafs. The tables contain the checksums of the file content, which can be used to retrieve the actual files from the Software Heritage web API. The files are separated into two categories:

- `content` (9.9M): Contains the checksum and the length of each file in the archive.
- `skipped_content` (5): Contains the checksum and the length of each file **not** archived.

Origins is not part of the DAG, it contains additional crawling information and states where and when a given snapshot has been captured.

- `origin_visit` (239k): Contains the different recurring visits of a given origin along with the date the visit occurred on. Points to the crawled `origin` and also to the `snapshot` table.
- `origin` (2.8k): The url of the project origin and its type (git, svn,...).

	File path	File size (hex)	Reference
k_1	/.gitignore	`00 01 DF 00`	1
k_2	/src/util/types.h	`00 01 02 E2`	2
k_3	/src/util/helpers.h	`00 02 11 AB`	3
k_4	/src/main.cpp	`00 02 CC 21`	4
k_5	/src/merger.h	`00 02 D2 C9`	5
k_6	/src/merger.cpp	`00 02 D2 CC`	6

Table 2.: File information for the example project from Table 1.

In this thesis we want to index the paths and the sizes of the files that are stored in the Software Heritage Archive. To achieve this we need the groups **revisions**, **directories** and **content**. The **revisions** point us to all root directories, which are the beginnings of each file path. Roughly 18% of the project revisions point to a duplicate project directory. Thus we deduplicate the *directory* attribute of the `revision` table to avoid calculating the same path/value pairs multiple times. **Directories** contains each file path split up in directory and file names, while **content** contains the file sizes. Table 2 shows the files of our example project from Chapter 1. Table 3 shows how the data of Table 1 is stored in the Software Heritage Archive. We can again see the usage of arrays for the directory- and file-entries. We also note that `directory_entry_dir` and `directory_entry_file` do not use hashes as their ids, but rather `64bit` Integer values.

id	date (timestamp + time zone)	directory	message	committer
482d19...	2020-03-12 14:02:22+02	8d392f...	Added project files.	1

(a) revision

id	dir_entries	file_entries
8d392f...	[1]	[1]
56d616...	[2]	[4,5,6]
a516ce...	null	[2,3]

(b) directory

id	target	name
1	56d616...	src
2	a516ce...	util

(c) directory_entry_dir

id	target	name
1	f9fa01...	.gitignore
2	609633...	types.h
3	be9d72...	.helpers.h
4	fc9eae...	main.cpp
5	84d1ba...	merger.cpp
6	99248f...	merger.h

(d) directory_entry_file

sha1	sha1_git	length
d671f6...	f9fa01...	122,624
aab548...	609633...	66,274
8e9672...	be9d72...	135,595
2cfab2...	fc9eae...	183,329
378ffc...	84d1ba...	185,033
5a72bc...	99248f...	185,036

(e) content

Table 3.: The files from Table 1 split up over the relevant relational tables in the Software Heritage Archive.

To construct the RCAS index we need the Software Heritage data in a (`path`, `value`, `reference`) format, as depicted in Table 2, where `reference` points to the `directory_entry_file` *id*. We achieve this by using the unique directory entry points to get the root directories and recursively combining them with `directory_entry_dir` and `directory_entry_file` to get the file paths. We then combine the file paths with the `content` to get the file sizes. We decide to not include the repository names in our file paths, as they only seem to appear as part of the `origin` *url*, which does not have a consistent format to easily extract the names.

2.2. SQL Solution

The approach outlined in Section 2.1 can be directly implemented in SQL. Figure 2 shows the code used to generate all the file paths, file sizes and their references. We define two common table expressions (CTE): The basecase joins all distinct root directories from `revision` with `directory` and then keeps track of all the files and the subdirectories. *curPath* is the so far accumulated file path. Each file path starts with a slash `/`. The recursive `path` CTE gets the names of all the subdirectories and appends them to the *curPath* attribute by joining with `directory_entry_dir`. We still need to join with `directory` in each iteration to get the next level of subdirectories and files. After concatenating all paths, we join our `path` CTE

```sql
WITH RECURSIVE
basecase AS (
  SELECT dir_entries, file_entries, CONCAT('/') as curPath
  FROM (SELECT DISTINCT directory FROM revision) as r
  JOIN directory d ON r.directory = d.id
),
path AS (
  SELECT *
  FROM basecase
  UNION ALL
  SELECT dir.dir_entries, dir.file_entries, CONCAT(path.curPath, name,
  ↪  '/')
  FROM path
  JOIN directory_entry_dir ded ON ded.id = any(path.dir_entries)
  JOIN directory dir ON ded.target = dir.id
)
SELECT DISTINCT concat(curPath, name) as filepath, length, def.id
FROM path
JOIN directory_entry_file def ON def.id = any(path.file_entries)
JOIN content ON def.target = content.sha1_git
ORDER BY filepath, length;
```

Figure 2: SQL query to compose all path/value pairs.

with `directory_entry_file` and `content` to retrieve all the file sizes. We need to use recursion for this query, as we do not know how many `join` iterations we will need due to the file paths not being known in advance. With the recursive CTEs we can repeat the join operation until no new results are added.

We experienced severe performance issues when we executed this SQL query on a subset of the `popular-3k-python` dataset. For 100 root directory references from the `revision` table (entry points) it takes 44 seconds (0.44s/entry point), for 1k it takes 2.4 minutes (0.15s/entry point) and for 10k 25.3 minutes (0.15s/entry point). This means for the complete `popular-3k-python` dataset with 4.3M unique entry points it would take around 5 days with an optimistic 0.1s per entry point. This is not feasible for the `popular-3k-python` dataset and also not for the complete Software Heritage Archive, which is magnitudes larger.

There are multiple reasons for the bad performance, we list three possible ones:

- The SQL query saves all intermediate results, due to the `UNION (ALL)` clause in the recursive CTE. If we look at Table 4, the left table displays a sample of the results computed from the `path` CTE in Figure 2 before we join it with the `directory_entry_file` table. We see that we need most results due to the *file_entries* attribute, but we could discard all intermediate results that hold `null` values for file entries. We would like to discard those results as soon as we encounter them to achieve a resulting set as seen in Table 4. Small-scale tests show that with a sample of 1k revision entry points we can have up to 20% of intermediate results with a *file_entries* value of `null`, which is suboptimal.

dir_entries	file_entries	curPath
[1,2,4,16]	[2,7]	/
[3,5,9,65]	null	/src/
[2,5,11,14]	[1,4,11]	/src/java/
null	[14,16,17]	/tests/

(a) Sample of some intermediate results after recursively computing all file paths

dir_entries	file_entries	curPath
[1,2,4,16]	[2,7]	/
[11,14,2,5]	[1,4,11]	/src/java/
null	[14,16,17]	/tests/

(b) Optimal sample after recursively computing all file paths

Table 4.: Intermediate results of recursive CTE, showing intermediate results.

- We did not optimize our postgresql instance for our query. This means that postgresql may write intermediate `join` results to disk earlier than needed due to the default `work_mem` settings, which would cause worse performance.
- We know that the Merkle DAG is fully deduplicated, this means that during our query we may often have multiple nodes that have the same child node. Here we would compute the same subtree each time it appears, instead of computing it just once for each unique file path.

To avoid the above mentioned problems we try to implement a recursive algorithm that works fully in-memory and also discards of any unneeded intermediate and duplicate results directly.

2.3. In-Memory Solution

Overview: In this approach we want to load all table data directly into memory and keep it there, avoiding costly disk read/writes of intermediate results. This means we have to be conservative with our limited memory available. We choose to implement this approach in Java. We create classes that hold the relevant table information in arrays in columnar table layout. We can then recursively create our file paths by following the pointers from our root directories to our files. Like the SQL approach in Section 2.2, we use recursion as we do not know beforehand how often we need to repeat our path joining.

To reduce the amount of data and tables that we need to read into our program, we decide to create temporary tables by removing unnecessary attributes from the Software Heritage tables. We can then easily dump our temporary tables as CSVs for the program to read. We create four temporary tables:

- `directory'`(*id, dir_entries, file_entries*)
- `directory_entry_dir'` `(DED)`(*id, target, name*)
- `directory_entry_file'` `(DEF)`(*id, name, length*)
- `entry_dirs`(*id*)

For `directory'` and `DED` we remove any additional attributes from their original tables. We join the `content` table with `DEF` to add the *length* attribute directly. `entry_dirs` contains the unique directory ids from the `revision` table. As all our later table searches will happen by *id*, we sort each table by its id. We transform the `bigint[]` arrays in `directory` by separating them with an empty space when exporting. This makes parsing them trivial compared to the default export format of arrays which uses "`{long1,long2}`". The *name* attributes are encoded as a hex string, thus we can safely use a comma as our attribute separator for exporting and parsing the created CSV files.

```
public class Ded {
  long[] id;
  byte[][] target; //reference to Directory id
  String[] name;
  int curIndex = 0;

  Ded(int numEntries){/* Initialize Arrays with size */}

  void insertRow(String line, Dir dirs){
    String[] lineArr = line.split(",");
    byte[] targetId = Dir.gitShaToByteArr(lineArr[1]);
    int targetIndex = Arrays.binarySearch(dirs.getId(), targetId,
     ↪  Arrays::compareUnsigned);

    id[curIndex] = Long.parseLong(lineArr[0]);
    target[curIndex] = dirs.id[targetIndex];
    //convert hex string to smaller byte array back to string
    name[curIndex] = (new
     ↪  String(Dir.gitShaToByteArr(lineArr[2]))).intern();
    curIndex++;
  }
}
```

Figure 3: Directory_entry_dir class layout.

Data Structures: The Software Heritage Archive stores the table ids either as a `sha1` hex string or `long` number. In the case of hex Strings we convert it to a byte array to cut the memory usage in half. A hex string "`\x737263`" would be saved as a byte array of size three with values `{73 72 63}`, using three bytes instead of 8 for the storage. As the *id* for `entry_dirs` is a `sha1` value, we use a 2D array to hold all the ids as starting points to create our file paths.

For the other tables we create a class each with an array for each attribute. We use only one class per table to avoid additional object overhead and byte-alignment. We use a column-based storage of attributes as most of our searches are only for one attribute (*id*), while in the SQL solution the database was row-store oriented. Figure 3 shows the class layout for `DED`: It uses a long[] to store all its ids, a 2D byte array for the references back to the `directory` id and a String array for the names. Each row of the attributes refers to a row in the CSV. In the case

of *id*, this array is automatically sorted due to how we exported our temporary tables, which means we can use a binary search to quickly find rows. The classes for `DEF` and `directory` have a similar structure.

Input Reading: We parse all CSVs by using the Java Stream API and a buffered file reader for faster reading. We then insert each row in the appropriate class. Figure 3 shows the insertion procedure for a row in the `DED` table. We split each row to get the *id, target* and *name* separately. Instead of saving the *target* directory id directly, we search for that directory entry in the Directory (Dir) class and save a reference to its id as *target*. This way we only need to save the id once in the Dir class and can refer to it with a pointer, saving memory. We also convert the *name* from a hex String back to ASCII to reduce its size. A hex string `\x737263` would lead to an ASCII string `src`, cutting the size in half.

String Interning: As many names, such as `src, tests, .gitignore` are repeated often, we decide to intern them. Interning a string results in it only being allocated once in the String Pool. When creating a new interned string that already exists inside the String Pool, it is not allocated again but refers to the existing string. This reduces the memory consumption, but increases the execution time slightly, as the program has to check the String Pool on each string creation.

Since JDK 7 HotSpot JVM, the interned String pool is no longer saved in the PermGen area but on the heap, allowing large amounts of Strings to be interned [1]. String interning saves less memory than expected, as our peak memory consumption happens during the path creation phase, where we create many intermediate strings that are not interned. However, the maximum needed memory is still lower with interning.

File Path Creation: The path creation algorithm can be seen in Figure 4. We start with the `directory` id given from `entry_dirs` and get the corresponding directory. The first `if` clause handles the recursive calls to all subdirectories, the second clause is for file handling. There we create a new `FileEntry` for each result and save it in our resulting set, which means duplicates get eliminated automatically.

As our dataset represents a deduplicated Merkle DAG, we often have different projects referring to the same directory. This means we have multiple calls on the same directory node, which leads us to create identical path/value pairs multiple times. Optimally we want to stop the path creation if for a given prefix `a` a directory node has already been visited, but continue if we visit the node with a different prefix `b`. This may occur when two projects have identical subfolders but with a different directory path. We achieve this check by using a HashMap with the `directoryIndex` (Figure 4, line 2) as a key and a list of prefixes as its values. We do this via the `handleDuplicates` method. This leads to faster completion times but occupies more memory. Depending on the given memory limitations it may not be a good choice.

Evaluation: The main advantage of the Java solution compared to the SQL solution is the execution speed, which is magnitudes faster. This comes with the restriction that the dataset is limited by the amount of available RAM. The CSVs from the database dump are around 6.8GB in size, the Java parser needs close to 30GB of RAM. This is roughly a factor 5 increase in memory needed and mostly comes from the `calculatePath` function in Figure

```
void calculatePath(HashSet<FileEntry> result, byte[] id, String curPath){
    int dirIndex = Arrays.binarySearch(dirs.getId(), id,
    ↪  Arrays::compareUnsigned);

    handleDuplicates(dirIndex);

    if(dirs.dir_entries[dirIndex] != null){
        for(long dedId:dirs.dir_entries[dirIndex]){
            int curDedIdx = Arrays.binarySearch(deds.id, dedId);
            calculatePath(result, deds.target[curDedIdx],
            ↪  curPath+deds.name[curDedIdx]+"/");
        }
    }

    if(dirs.file_entries[dirIndex] != null){
        for(long fileId:dirs.file_entries[dirIndex]){
            int curDefIdx = Arrays.binarySearch(defs.id, fileId);
            if(curDefIdx < 0){ break; } //if file was skipped_content
            var fileEntry = new FileEntry(/*params */);
            result.add(fileEntry);
        }
    }
}
```

Figure 4: Recursive function to create all Path/Value/ID triplets.

4 because we have all the partial `curPath` strings and also our results saved in a memory-inefficient `Set`. Due to the high memory consumption this approach is unlikely to work for the whole Software Heritage Archive with its size of 1.2TB - 250 times larger than the `popular-3k-python` used here.

3. RCAS Index

In this chapter we give an overview of Content and Structure interleaving and how we can use different attribute interleaving techniques for more efficient query execution. We describe our design choices for our RCAS index implementation and show how we implemented various algorithms.

3.1. Content and Structure Indexing

As CAS queries affect both the content and the structure of the data, the question arises, if we should handle the two query parts sequentially or in parallel via interleaving. This has a big impact on the index structure and query performance. For example if we were to sequentially execute first the content and then the structure query or vice versa, we would often have huge intermediate results. For Q_2: `//src//, size<500` from Chapter 1, if we first execute the content query we would have all files with a size smaller than 500 bytes and then would need to filter out which of them meet the structure requirement. For big datasets loading and processing all those intermediate results is not a good option.

Contrary to other index implementations for CAS queries, which often build separate indexes for content and structure or prioritise one dimension, the RCAS index is well-balanced and offers a robust performance [9]. This also means that we reduce the problem of having large intermediate results, as with the dynamic interleaving of the dimensions we query both at the same time.

While the RCAS index is designed for any type of CAS queries, the implementation of this thesis focuses on the combination of path and value dimensions with path being a full file path and value being the size of a given file. We support simple searches for file ranges such as $Q_C = 100 \leq size \leq 1000$ for content, as well as searching for a given path. Additionally we implement the descendant axis `//`, which matches zero to any number of descendants. For example a path query $Q_P =$`/a/b//` would match `/a/b`, as well as `/a/b/desc1/desc2`. We also implement the wildcard character $\star$, which can be used to match any path between two slashes. Using the Software Heritage Archive as an example, it could be used to match all files directly located in the test folder with the query $Q_P =$`/test/`$\star$, or to skip one folder hierarchy to get all files in the `include` folders which could be located in both `/test/include` and `/src/include`. We could achieve this with a query such as $Q_P =$`/`$\star$`/include//`.

3.2. Dynamic Interleaving and RCAS Structure

An important property of the Robust Content-and-Structure (RCAS) Index is its dynamic interleaving of the path and value attributes for each entry. We refer to each pair of PATH/VALUE attributes as a composite key `k(P,V)`, where P is the file path and V is the file size displayed as a `32bit` unsigned Integer. The following chapter is based on [9]. We use the same notation as [9] for composite key sets: $K^{1..6}$ refers to $\{k_1, k_2, k_3, k_4, k_5, k_6\}$ and $K^{2,5,6}$ refers to $\{k_2, k_5, k_6\}$.

Consider the two composite keys k_a: `/a/code/xy.z, 00 01 B9 5F` and k_b: `/a/code/z.y, 00 01 CC DF`. Using concatenation we can join the dimensions together as either `PV` or `VP`. This has the disadvantage that in the case of `PV`, we have to read the whole PATH until we can discard keys based on their VALUE predicate. In the case of a high PATH selectivity[1] but low VALUE selectivity, this would be very inefficient. The same problem exists for the `VP` concatenation. In both cases, any sort of interleaving of the two dimensions would speed up the selection process, as we can narrow down the relevant keys faster. If we decide to interleave each byte of the dimensions, we would get for example interleaving I_{ka} =`/00a01/B9c5Fode/xy.z`. This approach works well if we have dimensions with similar length. In the case where one dimension has significantly more bytes, the above approach prioritizes the shorter dimension, making it an inefficient interleaving. However, the dynamic interleaving approach proposes a better way to interleave the bytes. We look at all the keys and interleave only at the first byte in each dimension where the keys differ (called discriminative byte), grouping them in the process. We repeat this until no discriminative byte can be found any more. This allows the dimensions to be more evenly interleaved and looks as follows: Key k_a differs from k_b at the following byte per dimension highlighted in bold: k_a: `/a/code/xy.z, 00 01 B9 5F`, which leads to an interleaving of I'_{ka} =`/a/code/00 01xy.zB9 5F`. Compared to I_{ka}, where we had 8 path bytes at the end, we now have two value bytes at the end of I'_{ka}, spreading the value bytes more evenly. To use this interleaving, we first need to define the terms *discriminative byte* and *ψ-partitioning*.

Discriminative Byte: The discriminative byte $dsc(K, D)$ of a set of composite keys K in dimension $D \in \{P, V\}$ is the position of the first byte in dimension D for which not all keys are equal. If all values of dimension D are equal, the discriminative byte does not exist. In this case we set the discriminative byte to the length of dimension $D + 1$ of any key in K.

Composite Keys K	dsc(K,P)	dsc(K,V)
$K^{1..6}$	2	2
$K^{3..6}$	6	3
$K^{5,6}$	13	4
K^{6}	16	5

Table 5.: Discriminative Byte distribution for the keys from Table 2 and various subsets.

[1] We define the selectivity as the percentage of data returned.

Table 5 shows the positions of the discriminative bytes for the PATH and VALUE dimensions on the keys from Table 2. To group the composite keys that have the same value for the discriminative byte in dimension D together, we have to partition the set of keys K. This means we can have at most 256 different partitions for a given discriminative byte, one for each possible byte value.

ψ-Partitioning: $\psi(K, D) = \{K_1, ..., K_m\}$ is the ψ-partitioning of composite keys K in dimension D if and only if:

- All partitions are non-empty
- The number m of partitions is minimal
- All keys in partition $K_i \in \psi(K, D)$ have the same value for $dsc(K, D)$
- The partitions are disjoint
- The partitioning is complete, each key is in a partition $K_i \in \psi(K, D)$

We show the ψ-partitioning for selected sets of our composite Keys in dimension P or V from Table 2:

- $\psi(K^{1..6}, P) = \{K^1, K^{2..6}\}$
- $\psi(K^{3..6}, V) = \{K^3, K^4, K^{5,6}\}$
- $\psi(K^{3..6}, P) = \{K^3, K^{4..6}\}$
- $\psi(K^6, P) = \psi(K^6, V) = \{K^6\}$

Interleaving: In order to dynamically interleave our keys, we recursively ψ-partition our set of keys. After each iteration we alternate the dimension which we partition in. If this is not possible, we use the same dimension as the previous iteration. This happens when all values in a partition are equal in the given dimension. If we cannot ψ-partition a set of keys any further in any dimension we stop and assign it the leaf dimension $\perp$. For each of the partitioned sets we assign path and value substrings $s_P = k.P[dsc(K_{i-1}, P), dsc(K_i, P) - 1]$ and $s_V = k.V[dsc(K_{i-1}, V), dsc(K_i, V)]$. $k.P$ denotes the path of a key k in the given partition and the interval is the byte sequence between the previous and the current discriminative byte, but not including the current discriminative byte. Consider a key $k =$`/hello/world, 20 01 02 DF`, a previous discriminative byte position of 2 and a current one of 4 for both path and value dimensions. We then have $s_P = k.P[2, 4 - 1] =$`he` and $s_V = k.V[2, 4 - 1] =$`01 02`. Table 6 shows the dynamic interleaving for each key from our example project in Table 2. Each tuple contains s_P, s_V and dimension D. Here we started our interleaving with the VALUE dimension.

Key	Dynamic Interleaving $I_{DY}(k, K^{1..6})$
k_1	(/, 00, V), (ϵ, **01**, P), (.gitingore, DF 00, $\perp$)
k_2	(/, 00, V), (ϵ, **01**, P), (**s**rc/util/types.h, 02 E2, $\perp$)
k_3	(/, 00, V), (src/, **02**, P), (**u**til/helper.h, 11 AB, $\perp$)
k_4	(/, 00, V), (src/, **02**, P), (**m**, ϵ, V), (ain.cpp, **CC** 21, $\perp$)
k_5	(/, 00, V), (src/, **02**, P), (**m**, ϵ, V), (erger., **D2**, P), (**h**, C9, $\perp$)
k_6	(/, 00, V), (src/, **02**, P), (**m**, ϵ, V), (erger., **D2**, P), (**c**pp, CC, $\perp$)

Table 6.: Dynamic Interleaving of the composite Keys from Table 2. Discriminative bytes are displayed in bold.

RCAS Index Structure: We implement the RCAS index as an Adaptive Radix Tree [2]. We use 4 intermediate node types with sizes of 4, 16, 48 and 256 to refer to its children. Each intermediate node also keeps track of its dimension D, a path substring s_P and a value substring s_V. We use different sizes for intermediate nodes depending on how many child nodes they have, to reduce the memory consumption. A leaf node has a similar structure to an intermediate node, but instead of children it holds the references of our keys. Figure 5 shows the resulting RCAS index using the dynamically interleaved composite keys from Table 6. The discriminative bytes are highlighted in bold. Each leaf node has a set of references ($\{r\}$) that point to file ids.

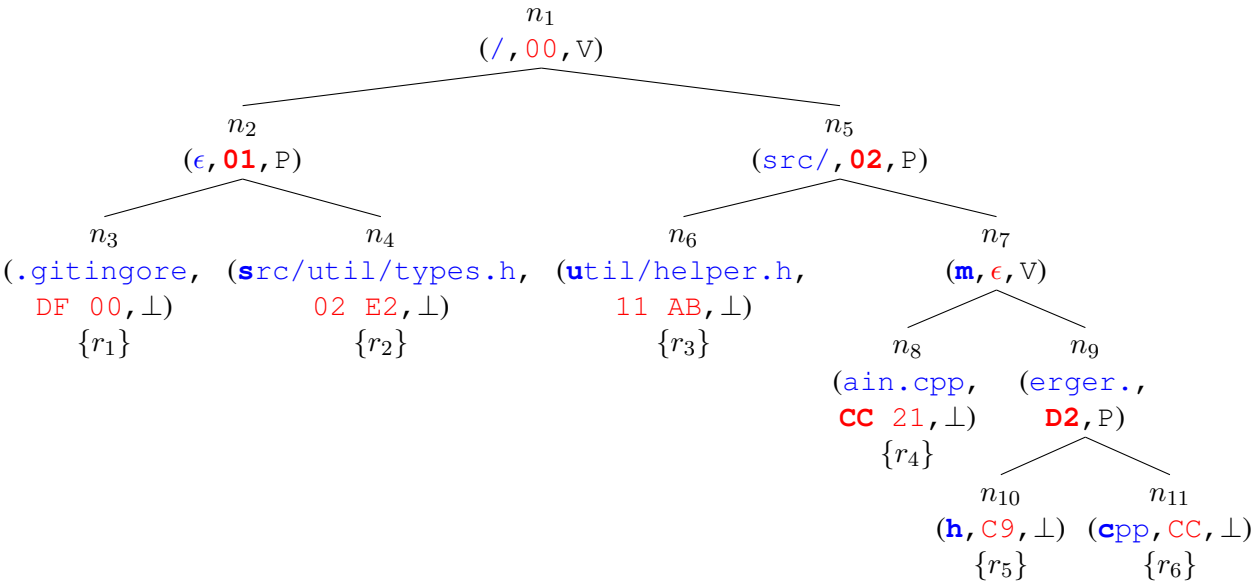

Figure 5: RCAS index of the composite keys from Table 2 and their dynamic interleaving in Table 6.

3.3. RCAS Implementation

3.3.1. Language and Data Types

To implement the RCAS index we choose the C++ programming language, because it offers both high-level abstractions but still allows detailed memory management. We choose C++17 to allow access to more modern features and simpler syntax, such as range-based for loops.

We define several types to use in our implementation:

- `using ref_t = uint64_t`, which is used for the references in each leaf. `uint64_t` refers to an unsigned integer with a size of 64bit (8byte). We create a custom type instead of using an unsigned integer directly to allow easy change of the type if necessary. We use `ref_t` to refer to the ID of a `directory_entry_file` entry in the Software Heritage Archive.
- `using PV_key_t = pair<vector<uint8_t>, vector<uint8_t>>`, which holds a pair of byte vectors consisting of a path and a binary comparable value byte vector. To make a signed integer binary comparable we have to flip the first bit, as it represents the sign, which is `1` for negative numbers and `0` for positive numbers [2]. Section 3.3.2 gives more detail on how to create a binary comparable value byte vector. We use a `pair` data type from the standard library as an easy way to hold the two vectors together.
- `using keyList_t = list<pair<PV_key_t, ref_t>>` is used to create a list of all keys, which are a pair of `PV_key_t` and a `ref_t`. We can use a list instead of a vector as we only need to traverse it from front to back and also append elements to its end. Unlike a vector, a list never needs to resize when multiple items are appended.

We also define `enums` for the dimensions of a node, the node type and for certain matching conditions in the query algorithm. We set `enum-base` to an `uint8_t` instead of the default 4 bytes that it uses. This gives us a greater readability compared to using an `8bit` integer directly, while requiring the same amount of memory.

```
1  class node_t {
2  public:
3      dim d_;
4      node_type n_type_;
5      vector<uint8_t> sp;
6      vector<uint8_t> sv;
7
8      virtual ~node_t();
9  };
```

```
class nodeInner_t: public node_t {
public:
    int16_t num_children;

    virtual void insert_node(node_t *n,
     ↪  uint8_t keyByte) = 0;
    virtual node_t** get_child_pointers()
     ↪  = 0;
};
```

Figure 6: Header files with the node_t and nodeInner_t class declarations.

We structure our nodes using class inheritance, see Figure 7. Figure 6 shows the sample class declaration for classes `node_t` and `nodeInner_t`. Our parent class `node_t` holds

the node type and dimension `enums` as well as the s_P and s_V vectors. We also declare a virtual destructor such that we can delete the derived classes through a base class pointer. The `nodeLeaf_t` and `nodeInner_t` classes both inherit directly from `node_t`. the leaf node adds a reference vector and the inner node adds a counter for the amount of children it has. Additionally we define intermediate nodes with specific sizes from 4 to 256 that inherit from `nodeInner_t`. Each of those intermediate nodes has an array with pointers to its child-nodes. The nodes with sizes 4, 16, 48 additionally have a key array for more efficient access to the child nodes. We refer to [2] for a more in-depth explanation on the keys and children implementation.

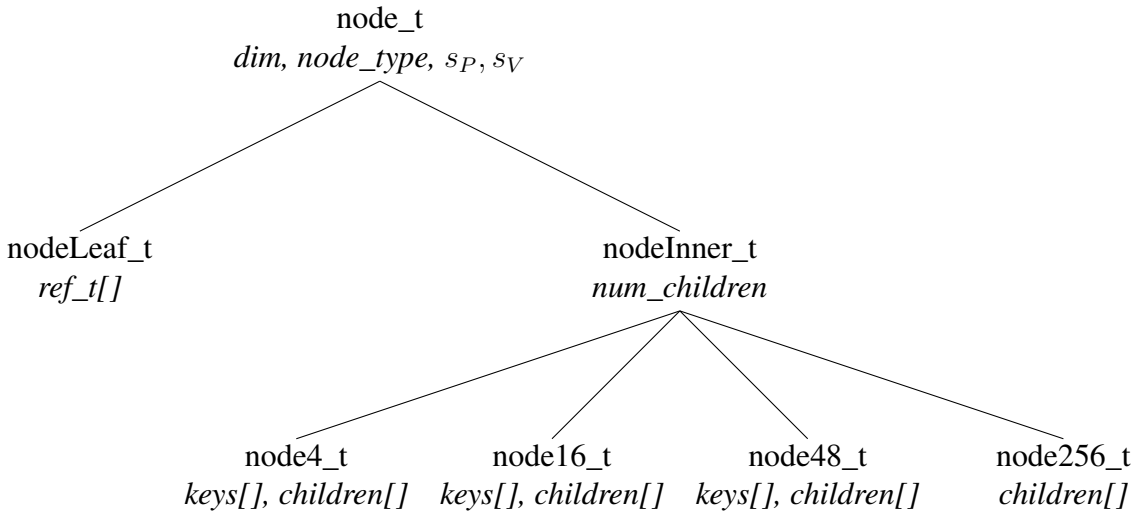

Figure 7: Node inheritance structure.

3.3.2. Algorithms

To create the RCAS index, we need to have a complete list of the composite keys and their reference value `ref_t`.

Data Parsing: While parsing the resulting CSV from our In-Memory solution in Section 2.3, we can create the path byte vector s_P by simply iterating over every PATH character and pushing it into the vector. For the value vector s_V we need to take into account its binary comparability and also the memory endianness. For our implementation we want the value bytes saved in big endian. To check if we are on a big endian machine, we can create a two byte number, get the first byte of it and check its content. As can be seen in Figure 8, we save the number 1 as two bytes, `0x00` and `0x01`. We then return the first byte in order via a `reinterpret_cast` and see if it equals `0x00` or `0x01`. We need binary comparability as we want to compare our values by looking at each individual byte. To achieve that we need to flip the first bit of each number due to it being the sign bit, which determines if the number is negative or positive. This is due to negative numbers being represented as a two's complement. If we do not flip it, we would classify all negative numbers to be larger than

all positive numbers. Lastly, to push each individual byte into the value vector we can left shift[2] our number in a big endian system such that the desired byte is at front and use a similar approach as in Figure 8 to get the value of the byte and save it. If we encounter a little endian system we just read the number byte-wise from back to front to get the correct order. The whole process for a big endian system can be seen in Figure 9.

```
bool isBigEndian(){
    int16_t test = 0x0001;
    const bool bigEndian = *reinterpret_cast<char*>(&test) !=1;
    return bigEndian;
}
```

Figure 8: Determining endianness of the current system at runtime.

```
void int_to_bitwise_bin_comp(int64_t value, vector<uint8_t> &byteVec){
    uint64_t XOR_mask=1;
    XOR_mask <<= 63;
    value = value^XOR_mask;
    for(uint16_t i=0; i<sizeof(value)-1; ++i){
        auto tempValue = value<<(i*8);
        byteVec.push_back(*reinterpret_cast<uint8_t*>(&tempValue));
    }
}
```

Figure 9: Converting a 64bit signed integer to a binary comparable byte vector on a big endian system.

Discriminative Byte: We evaluate the discriminative bytes for both the path and value dimensions with the simplified `dsc_inc` function in Figure 10. We take the first key of the set, and from the position of the last discriminative byte (`diffPos`) to the end of the current key, we check if any other key has a different character than the one of the first key. As soon as we find a different one, we return this new position, else we return one past the length, as given in the definition from Section 3.2.

ψ-Partitioning: In Figure 11 we show the ψ-partitioning for the PATH and VALUE dimensions. We have a `partitions` array of size 256. We then iterate over each key, get its character value at the position of its discriminative byte and use that to determine the index of the `partitions` array where the key will be inserted. We use `std::move` to indicate that the value may be moved efficiently instead of just copying it.

Figure 12 shows a simplified version of the RCAS construction algorithm. We use DIM to refer to the current dimension, e.g. `g_DIM_new = g_p_new` if we are in the PATH dimension.

[2]Bit shifting a value by x bits to the left (operator «) means it moves it by x bits to the left while adding zeros, e.g. `1111 1001«4` equals `1001 0000`

```
uint16_t dsc_inc(keyList_t &keys, dim &DIM, uint16_t diffPos) {
    //DIM = PATH or VALUE
    auto* k.DIM = &keys.front().first.DIM
    while(diffPos < k.DIM->size()){
        for(auto &key:keys){
            auto currentByte = key.first.DIM[diffPos];
            if(currentByte != (*k.DIM)[diffPos]){
                return diffPos;
            }
        }
        diffPos++;
    }
    return diffPos;
}
```

Figure 10: Simplified function to determine the discriminative byte for a given dimension and set of keys.

```
void psi_partition(keyList_t &keys, dim &DIM, uint16_t &diffPos, keyList_t
↪  *partitions) {
    //DIM = PATH or VALUE
    for(auto &key:keys){
        partitions[key.first.DIM[diffPos]].push_back(std::move(key));
    }
}
```

Figure 11: Psi-partitioning for PATH and VALUE dimensions.

To create the actual index via our `constructRCAS` function, we take the set of keys and determine the discriminative byte. We then assess if the node to be created is a leaf node or an intermediate node. We have a leaf node if both discriminative byte positions are larger than their path and value length respectively. In that case we can fill in the s_P and s_V values and set the dimension to LEAF. We then copy the references into the node and return the node, as can be seen in the `if` clause in Figure 12, line 7.

If it is not a leaf node, we have to check if the discriminative byte of the current dimension D is smaller than the length of D, in order to do a ψ-partitioning in D. Else we would need to do it in $\bar{D}$. We ψ-partition the set of keys to determine how many children the node will have. We give the intermediate node the smallest possible size based on the number of partitions. Finally we go through each set of keys produced by the partitioning and recursively call the `constructRCAS` function with $\bar{D}$ to create the child nodes.

Querying RCAS Index: A query traverses the RCAS index in a depth-first fashion and excludes subtrees that do not need to be traversed. In order to query the index, we define a query object that executes the query for us. It is initialized with a given query and holds the resulting set of references. We use an `enum` with values `MATCH, MISMATCH` to state whether a node matches or mismatches in the path or value dimension. We use `INCOMPLETE`

```
node_t* constructRCAS(keyList_t &keys, dim DIM, uint16_t g_p, uint16_t
↪  g_v) {
    auto* firstKey = &keys.front().first;

    uint16_t g_p_new = dsc_inc(keys, dim::PATH, g_p);
    uint16_t g_v_new = dsc_inc(keys, dim::VALUE, g_v);

    if(g_p_new >= firstKey->first.size() && g_v_new >=
    ↪  firstKey->second.size()){
        auto* leaf_node = new nodeLeaf_t();

        addAttributes(leaf_node, keys, g_p_new, g_v_new);
        return leaf_node;
    }

    if(g_DIM_new >= firstKey->DIM.size){
        invertDIMValue(DIM);
    }

    auto *partitions = new keyList_t[256];
    psi_partition(keys, DIM, g_DIM_new, partitions);

    nodeInner_t* inner_node = constructInnerNode(getNodesize(partitions));
    addAttributes(leaf_node, keys, g_p_new, g_v_new, DIM);

    for(int b=0; b<256; ++b){
        if(!partitions[b].empty()){
            inner_node->insert_node(constructRCAS(partitions[b],
            ↪  invert_dim(DIM), gp_new, gv_new), b);
        }
    }
    delete[] partitions;
}
```

Figure 12: Simplified RCAS construction algorithm.

when we cannot make that decision yet, due to needing more information. We first give a detailed explanation of the three main parts of the query function and then describe how they work together.

Query Buffers: We create a byte vector (buffer) for the PATH and VALUE dimension. We use these buffers to keep track of the so far encountered path and value bytes. For each intermediate node we visit we update the buffers, which are default initialized, by appending s_P and s_V to them. As mentioned in [9], we pass those buffers by value each time we visit a new node, creating a copy. As we show in Section 4.3, this and the default initialization cause large performance problems for queries that traverse many nodes. We fix our implementation by initializing our buffers with a large default size as was mentioned in [9]. Additionally, we now only use one buffer for each dimension globally, overwriting the bytes in them as needed. We use a variable for each buffer to keep track of the insertion position. Updating the path and

value buffers now works by adding the s_V and s_P values of the new node to their respective buffers via `std::copy` at the specified position and updating the insertion position.

Value Matching: In Figure 13 we show a simplified version of the `matchValue` algorithm, which compares the value range of the query with the value byte buffer. It uses the value buffer and an additional query state object, which holds the current positions in the buffers, the query path and the insertion positions for the buffers. `matchValue` first checks if any byte of the so far encountered value buffer is outside of the query value byte range with the first two if statements. If none are outside the value range and we encounter a leaf node, we can return `MATCH` if the whole value buffer was compared to the upper and lower bound and no violation occurred. If we did not yet check the whole value buffer but can already tell that the value is within the bounds via binary comparability, we also return `MATCH`. An example for this in decimal would be the number 2334 and the bounds $[1000, 4000]$. Just by looking at the thousands number we see that `2` is between `1` and `4`. We then already know that 2334 is in our bounds without having to check the other decimals. Binary comparability does the same thing but looks at the byte values instead. We do the same in-bounds check for non-leaf nodes. If the value buffer does not yet contain the whole value and we currently only know that the value is on one of the boundaries, we have to return `INCOMPLETE`, as the value could still be out of bounds.

```
matcher matchValue(vector<uint8_t> &buffV, node_t *n, qState &s) {
    if(smallerThanLowerBound(vLow, buffV, s)){
        return MISMATCH;
    }
    if(largerThanUpperBound(vHigh, buffV, s)){
        return MISMATCH;
    }

    if(n->n_type() == leaf){
        if(s.vLo == buffV.size() || s.vHi == buffV.size()){
            return MATCH;
        }
        if(buffV[s.vLo] > vLow[s.vLo] && buffV[s.vHi] < vHigh[s.vHi]){
            return MATCH;
        }
    }
    else if(betweenLowHigh(buffV, vLow, vHigh, s)){
        return MATCH;
    }
    return INCOMPLETE;
}
```

Figure 13: Simplified matchValue function.

Path Matching: We also take a closer look at the `matchPath` function from Figure 14. At first we compare each character of the so far encountered path (`buffP`) with the query path until we either get a mismatch or arrive at the end of the buffer. If we arrive at the end

of the buffer at row 16, we check if both query and buffer matched up completely, which leads to a `MATCH`. If we reached the end of the file path but not the end of the query, we return a `MISMATCH`. Else we cannot make a decision yet and return `INCOMPLETE`. If we get a character mismatch while comparing the `buffP` with the query, we have the following scenarios:

- We encounter a descendant axis, depicted in the query with a `^`. We first check if the descendant axis is at a valid position. If it is at a valid position, we set a flag that we encountered a descendant axis in this query and note its position. We then continue comparing `buffP` with the query.
- We encounter a wildcard character `*`. This means we want to skip to the next `/` in our path. We traverse our path until we find the next `/` and continue matching as before, or we reach the end of `buffP`. If we reached the end of the file path and the end of the query, we `MATCH`. If the query is not finished, we `MISMATCH`. If neither query nor file path is finished, we have to return `INCOMPLETE` and continue.
- We have a normal character mismatch but have encountered a descendant axis before. In this case, we reset query and value buffer positions back to where the descendant axis was encountered. We then increase our `buffP` position to the next `/` in the path and continue matching from there. If no `/` can be found in the file path, we return a `MISMATCH`. If no `/` can be found in `buffP`, but we are not at the end of the file path, we can continue searching and return `INCOMPLETE`.
- If none of the above happens, we simply return `MISMATCH`.

```
matcher matchPath(vector<uint8_t> &buffP, qState &s) {
    while(s.curPosP < buffP.size()){
        if(qPath[s.curPosQ] == buffP[s.curPosP]){ s.incPos(); }
        else if(qPath[s.curPosQ] == '^'){
            handleDescendantAxis(buffP, qPath, s);
        }
        else if(qPath[s.curPosQ] == '*'){
            handleWildCard(buffP, qPath, s);
        }
        else if(s.descPosQ != -1){
            resetToDescAxis(buffP, qPath, s);
            continueToNextAxis(buffP, s);
        }
        else{ return MISMATCH; }
    }
    if(s.curPosP == buffP.size() && s.curPosQ == qPath.size()){
        return MATCH;
    }
    if(!buffP.empty() && buffP.back() == '\0'){
        return MISMATCH;
        }
    return INCOMPLETE;
}
```

Figure 14: Simplified matchPath function.

Query Evaluation: Figure 15 shows the `evaluateQuery` function, which is called on the root node of the RCAS index. We pass the query state object by value to have separate values for each node it is invoked on. We first update the path and value buffers with the so far encountered path and value bytes. We then compare the value buffer with the value range given from the query in `matchValue`. This can lead to three possible outcomes: The buffer value matches the value range (`MATCH`), we cannot make a decision yet if the whole buffer value will match or not (`INCOMPLETE`) or the encountered value is outside of the value range (`MISMATCH`). Due to the binary comparability we can also determine if a value matches or mismatches without having loaded the whole value in the buffer. This allows for faster invocation of the `collection` algorithm, which we will look at later. We then do the same evaluation on the path dimension with `matchPath`. Here, adding additional functionality such as a descendant axis or a wildcard character does not change the basic premise of our three return values, it just makes the evaluation more complex. If the value or path matching returned `MISMATCH`, we can return and continue searching a different node/subtree. If both checks return `MATCH`, we can invoke the collection algorithm on the current node, which collects all the references that are in the subtree rooted at the current node. Else, one of the checks returned `INCOMPLETE` and we have to traverse the subtree, which we do by recursively calling our query on the next node.

```
void evaluateQuery(node_t *n, vector<uint8_t> &buffV, vector<uint8_t>
↪  &buffP, qState s) {
    updateBuffers(n, buffV, buffP, s);

    matcher matchV = matchValue(buffV, n, s);
    if(matchV == matcher::MISMATCH){ return; }

    matcher matchP = matchPath(buffP, s);
    if(matchP == matcher::MISMATCH){ return; }

    if(matchV == matcher::MATCH && matchP == matcher::MATCH){
        CAS_Query::collect(n);
        return;
    }

    auto* inner_n = dynamic_cast<nodeInner_t*>(n);
    auto* child_nodes = inner_n->get_child_pointers();

    for(int i=0; i<getNodeSize(n); i++){
        if(*(child_nodes+i)){
            evaluateQuery(*(child_nodes+i), buffV, buffP, s);
        }
    }
}
```

Figure 15: Simplified query evaluation function.

4. Query Performance Evaluation

In this chapter we give an overview of our experimental setup and the dataset used for it. We then evaluate the performance of several sample queries.

4.1. Setup

The evaluation of the queries was conducted on a Windows 10 machine with an Intel i7-3770s 4 core processor, a 1TB 7200rpm HDD and 2x8GB of DDR3-1600 RAM. Both the index and the queries were compiled using mingw-w64 with the flags -O3 and -m64. If not stated otherwise, each runtime measurement is the average of 50 runs.

We use the `popular-3k-python` dataset provided by the Software Heritage Archive, which consists of 3052 repositories from GitHub, Gitlab, PyPI and Debian that are tagged as being written in Python [5]. We then extract the composite keys and reference ids using the approach from Section 2.3 which we use to bulk-load the RCAS index.

4.2. Dataset and Index Structure

The dataset consists of rows with the format '"PATH",VALUE,ID', where PATH is the complete file path of each unique file, VALUE is the size of its file and ID is the id associated with the file. Special characters, such as double quotes, are escaped with an additional double quote. For the `popular-3k-python` dataset there are around 10 million keys, 4 million unique paths and 467 thousand unique values (file sizes). The file sizes are all between 0 and 104 million bytes. As seen in Figure 16, around 20% of the file sizes are below 1000 bytes, while the majority of the files are below 20'000 bytes. The dataset itself has a size of 572 megabyte. Each composite key is unique. Duplicate entries were removed during the CSV creation, as they also reference the same ID and add nothing to the index.

Parsing the CSV file and reading it into memory takes 35.8 seconds, constructing the index then takes 39.2 seconds for a total initial construction time of 75 seconds. After creating the index it has a size of 637 megabyte, which is an increase in size of about 11% compared to the CSV file. In total there are 13.9M nodes, most of which are leaf nodes, as seen in Figure 17. We also see that there are barely any nodes with size 48 and 256. If we look at the path and value distribution for the non-leaf nodes, we see that nodes with size 4, 16 and 48 are primarily path nodes, and the nodes with size 256 are primarily value nodes. This makes sense, as we have a smaller amount of possible different values per path character than we have for the value part. The index itself has a height of 95, which is a lot. In Appendix A.1 we can see

which paths generate such a large index height. Additionally, the average leaf node depth is 9, which means that a height of 95 is an outlier and the index in general is more modest in height.

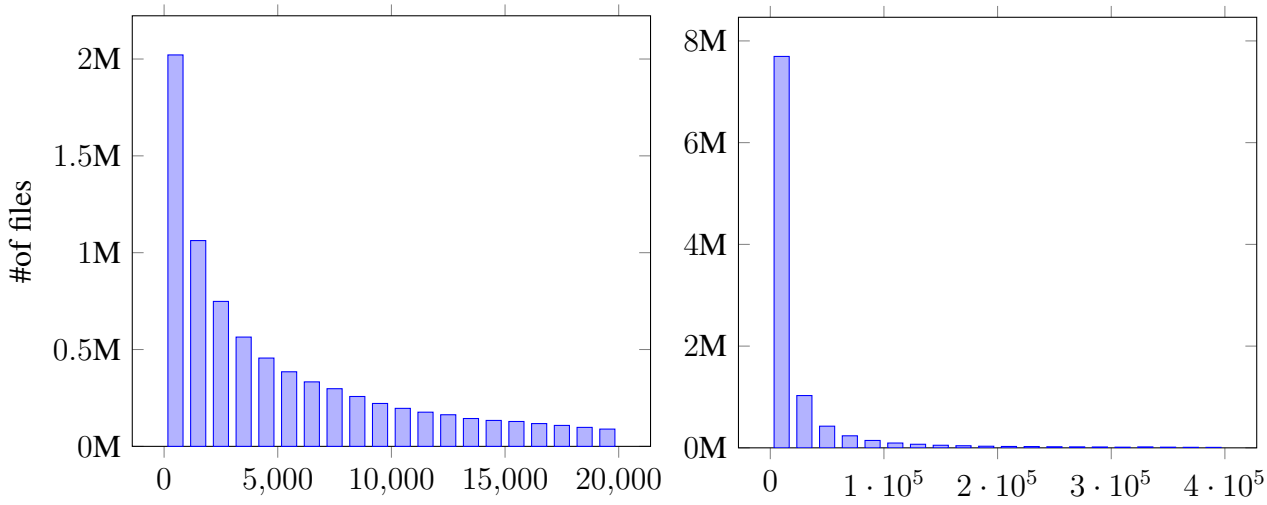

Figure 16: Distribution of the values (i.e., file sizes) in the ranges [0, 20k] and [0, 400k].

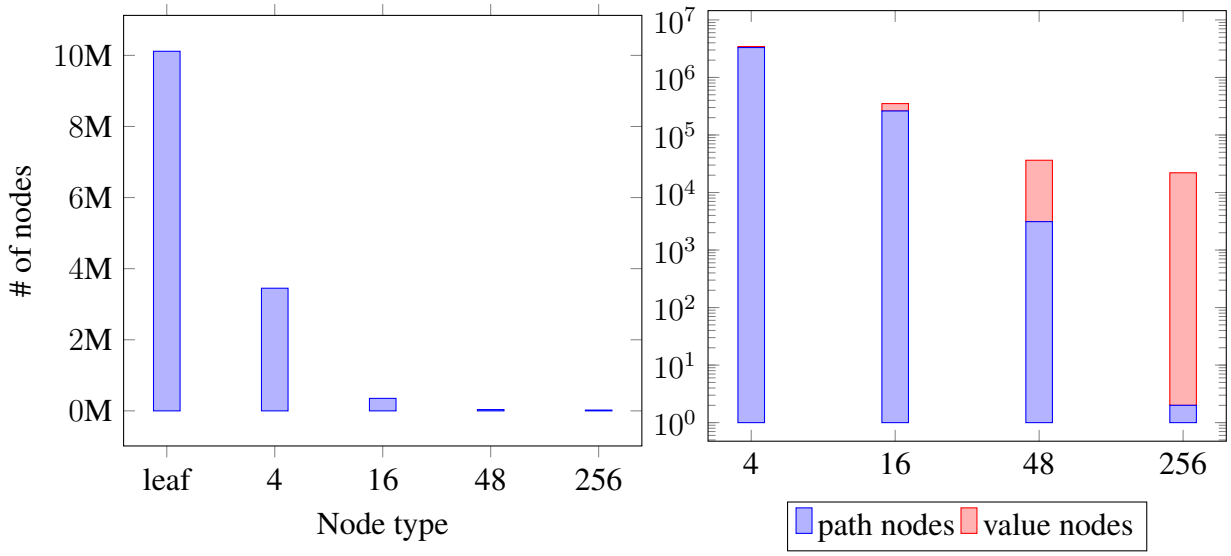

Figure 17: RCAS index node distribution and Path/Value distribution for given nodes.

4.3. CAS Queries Evaluation

To evaluate the performance of the wildcard character $\star$ and the descendant axis `//` for different situations we define the following queries as seen in Table 7. We report the number of results returned, the number of intermediate nodes that are traversed by the `evaluateQuery`

algorithm but do not yet invoke the `collect` algorithm and the number of nodes that are visited by the collect algorithm. We also state the percentage of the total keys returned and the percentage of total nodes returned.

Queries, $0 \leq size \leq 100k$		result size		trav. nodes		col. nodes	
Q_1	`/src/eyed3/utils/cli.py`	1	(<0.1%)	163,871	(1.2%)	1	(<0.1%)
Q_2	`//tests//`	1,191,911	(11.8%)	11,613,781	(83.1%)	1,409,668	(10.1%)
Q_3	`//tests/*`	366,203	(3.6%)	12,654,780	(90.6%)	367,136	(2.6%)
Q_4	`/*/include//`	30,893	(0.3%)	8,743,784	(62.6%)	37,604	(0.3%)
Q_5	`/src//nonexist`	0	(0%)	661,217	(4.7%)	0	(0%)
Q_6	`/src//`	486,498	(4.8%)	1,943	(<0.1%)	663,655	(4.7%)
Q_7	`/src/include//`	21,871	(0.2%)	173,932	(1.2%)	28,066	(0.2%)
Q_8	`/src/*`	34,435	(0.3%)	401,973	(2.9%)	37,604	(0.3%)

Table 7.: CAS queries with the number of results, the number of traversed and collected nodes.

The value selectivity for $size \in [0, 100k]$ is 0.94. Query Q_1 shows the performance of locating a file with a fully written out path. Q_2 returns all files with a `/tests` anywhere in their path. Q_3 is a subset of Q_2, only returning all the files directly in any `/tests` folder. Q_4 returns all files with `/include` located at the second hierarchy level, e.g. `/tests/include`. Q_5 searches for a non-existent file with the `/src` prefix. Queries Q_6 and Q_7 return all files where the path starts with `/src` or `/src/include` respectively, Q_7 returning a subset of Q_6. Query Q_8 returns all files that are directly in the folder `/src`, which is also a subset of Q_6. Each of the queries above is also executed for a $size \in [0, 5000]$ and $size \in [0, 1000]$, which have a value selectivity of 0.48 and 0.20 respectively, as can be seen in Table 8 of Appendix A.1.

The resulting query runtime is displayed in Figure 18. Looking at the runtime differences of each query separately, we see a clear difference between each value range, as is to be expected, due to the changing value selectivity.

As expected, providing a full path such as Q_1 results in the best performance. We also notice that the file size is a big factor for the query performance, despite there only being one valid file that matches the path. This is the case because if we use a large size range we cannot discard nodes based on the size and have to rely mostly on the path matching. We can see the increase of visited nodes for $Q_1, size \in [0, 1000]$, where we visit 15k nodes compared to $Q_1, size \in [0, 100k]$, where we visit 164k nodes - a factor 10 increase. If we know the exact size of the file (4217) we can further reduce the amount of visited nodes to 801 and the execution time to 0.9ms.

Queries that start with a descendant axis have the highest runtime and also visit the highest amount of nodes. This is the case because we cannot stop the node traversal early based on the path, as in the case of Q_2, there might be a `/tests` somewhere at the end of the path. Thus the only way to quickly restrict the amount of visited nodes is a limited value range. We also notice that Q_3 is slightly slower than Q_2. This happens because Q_2 can invoke the collection algorithm as soon as it finds a `/tests` in its path, while Q_3 has to continue traversing the subtree to see for further instances of `/tests` and can only collect the leaf nodes.

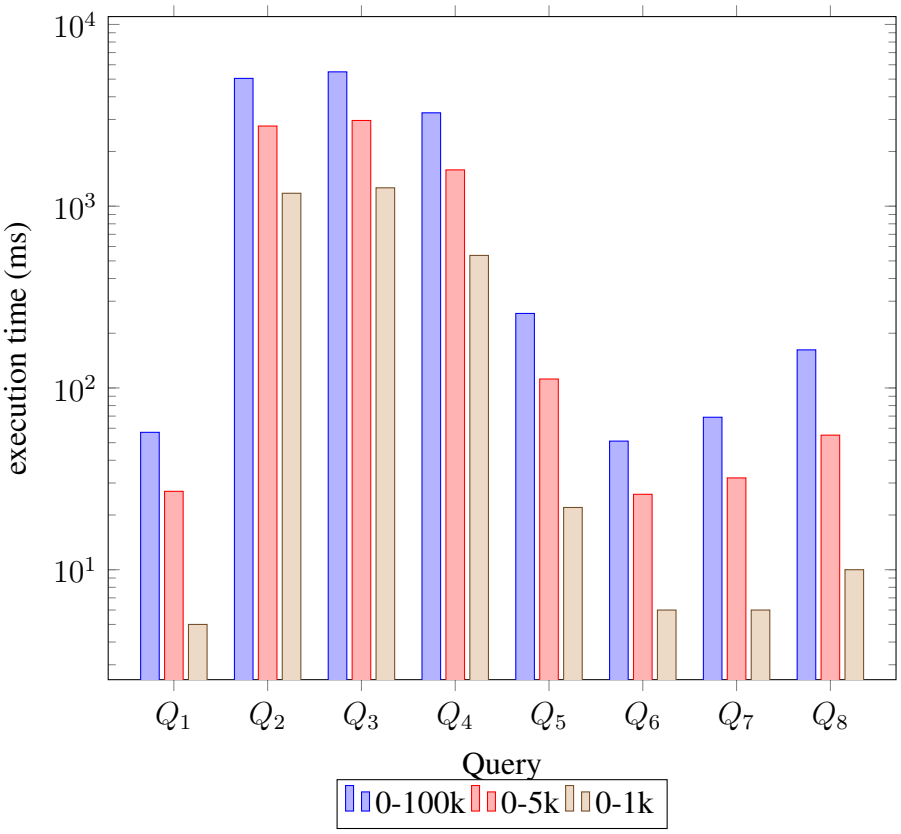

Figure 18: Runtime measurements for queries Q1 to Q8 for different file sizes using copied buffers.

Q_4 is in the same time range as the queries starting with a descendant axis. This is due to the wildcard character at the beginning of the query. It means that for each path we cannot discard the node until we encounter a second `/` and then a mismatch after. Thus having us traverse lots of nodes. If we compare with query Q_7, we see that having the wildcard character at the beginning leads to around 50% more results but also to a factor 50 increase in traversed nodes. This makes queries that start with a descendant axis or a wildcard character very expensive and should be avoided unless necessary.

For Q_5 we traverse roughly the same nodes as Q_6, with the only difference that in Q_5 we traverse all the nodes, while in Q_6 we are able to collect a majority of the nodes. Since Q_6 is faster than Q_5, we assume that the `collect` algorithm is faster than the traversal of the nodes.

If we compare query Q_6 and Q_7, we notice that even though Q_7 returns fewer results, visits fewer nodes and is a subset of Q_6, it takes longer to execute. Similarly with Q_8, which also returns a subset of the results of Q_6 but takes roughly double as long to execute.

Queries Q_5-Q_8 suggest that the `evaluateQuery` algorithm in Figure 15 is way slower than the collect algorithm. In order to test this, we analyse queries Q_6 and Q_8 from Table 7, as they either collect or traverse the majority of their nodes. We also analyse queries Q_2' and Q_3',

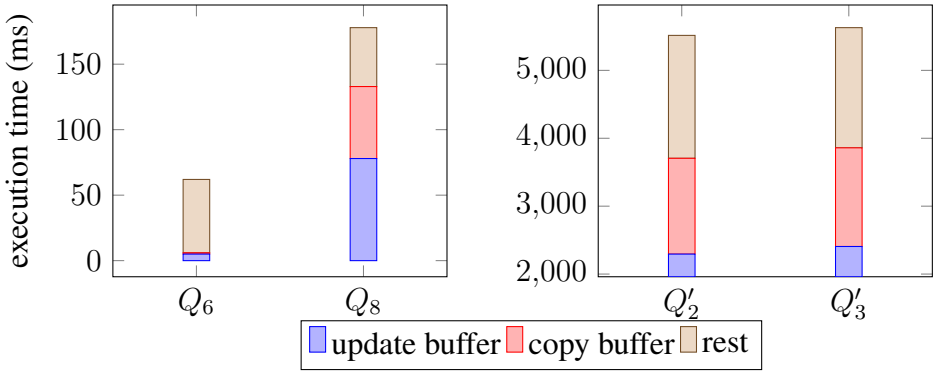

Figure 19: Buffer copy time in relation to query execution time.

which are a modified version of Q_2 and Q_3 with a $size \in [0, 900k]$. They both have a similar amount of traversed nodes, but a different amount of collected nodes. We measure the time it takes to copy the path and value buffers for each recursive execution call of `evaluateQuery` for each query and also the time spent in the `updateBuffers` function. As shown in Figure 19, we can see that there is only little time spent (9%) on updating and copying the buffers for Q_6, while for Q_8, 75% of the execution time is spent on buffer updating and copying. This difference can also be seen in the number of traversed nodes, Q_6 traverses roughly 2k nodes, Q_8 402k. Comparing Q'_2 and Q'_3, we see a similar amount of time spent on the buffers. This is also reflected in the amount of traversed nodes, roughly 12M for Q'_2 and 13M for Q'_3.

To reduce the update buffer time, we have to increase the initial size of the value- and pathbuffer vectors in order to avoid resizing. Currently the buffers are created initially empty with a default capacity. This means that when we update the buffers via `vector.insert`, we often trigger a reallocation of the entire vector. If we use a large buffer that can contain the whole path/value bytes without reallocating, updating the buffers is not a problem any more. But we still copy both buffers each time the `evaluateQuery` function is called. To avoid this, we can change the implementation to just use one path and one value buffer and pass those by reference instead of by value.

If we look at the results in Figure 20 now, we can see that queries, that solely relied on the `evaluateQuery` algorithm like Q_1, are now about 3x faster than before. Q_4, as well as the queries with a descendant axis prefix also gained a significant speed up, but still are considerably slower than the other queries. Query Q_5 is still slower than Q_6, which shows that traversing nodes is still more expensive than just collecting those nodes, but the time difference is not as large any more. Q_7 also executes roughly three times faster and is now as expected faster than Q_6.

To conclude we note that searching for a single known file is very fast (e.g. 0.9ms for query `/src/eyed3/utils/cli.py`, $size = 4217$). Additionally, due to the traversal of nodes being slower than the collection of nodes in this implementation, queries that end with a wildcard character are slightly slower or as fast compared to a query with an equal path and value, but having a descendant axis instead of a wildcard character at the end. This despite the wildcard query usually returning fewer results and visiting fewer nodes. Having a wildcard

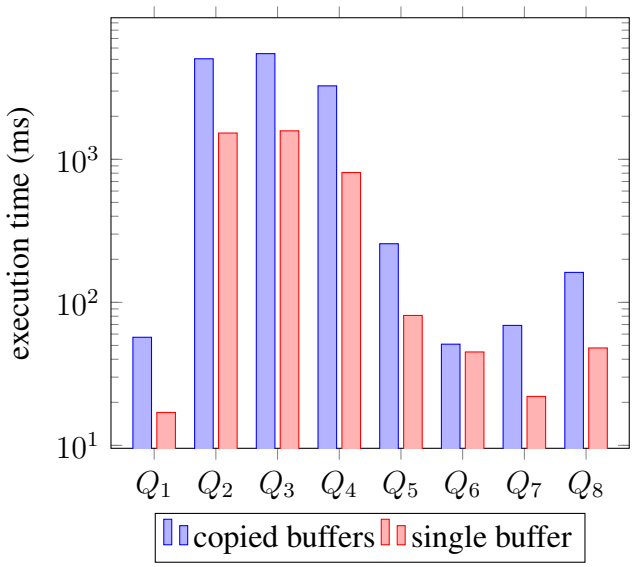

Figure 20: Runtimes for queries Q_1 to Q_8 with $size \in [0, 100k]$ with and without optimized buffers.

character or a descendant axis at the start of a query slows it down considerably, as seen in Figure 20.

5. Summary and Future Work

In this Thesis we integrated the RCAS index with the Software Heritage Archive to run CAS queries on the archive data and measure the performance of the index. We noted that the current access methods of the Archive were not sufficient for our needs. Thus we showed two approaches on how to parse the Software Heritage Archive to retrieve the requested file paths and sizes for our index implementation. After assessing the up- and downsides of each approach we concluded that both approaches would not be feasible for the whole Software Heritage Archive either due to time constraints (SQL approach) or due to memory constraints (In-Memory approach). Thus we used the superior In-Memory approach on a subset of the whole Archive.

We implemented the RCAS index and a query evaluator that also implements the descendant axis and the wildcard character for path matching. We tested the performance of our implementation on the `popular-3k-python` subset of the Software Heritage Archive, which was parsed using the In-Memory approach. Using the parsed subset we evaluated the general query performance and the performance impact of the descendant axis and the wildcard character at different query positions. During this we improved our query implementation by using two global buffers instead of copying them for each index node visited, which greatly increased the performance of queries that traversed many nodes. We concluded that queries with a descendant axis or wildcard character at the beginning of the path part of the query perform significantly worse than queries that use them at a later position, due to the amount of nodes that have to be visited.

Future Work lies in exploring an approach to efficiently parse the whole Software Heritage Archive and test the RCAS index on the extracted data. Multi-threading support for the query evaluator would also be a possible area to improve on.

A. Appendix

A.1. Additional CAS Query Information

Sample file path that generates an index of at least height 90, ˆ is placed after each new node starts:

```
/^^t^^e^^s^t^/^in^tegration/^target^s/^c^opy/files/subdir/^
    subdir^1/circles/^subdir^1/circles/^subdir^1/circles/^
    subdir^1/circles/^subdir^1/circles/^subdir^1/circles/^
    subdir^1/circles/^subdir^1/circles/^subdir^1/circles/^
    subdir^1/circles/^subdir^1/circles/^subdir^1/circles/^
    subdir^1/circles/^subdir^1/circles/^subdir^1/circles/^
    subdir^1/circles/^subdir^1/circles/^subdir^1/circles/^
    subdir^1/circles/^subdir^1/circles/^subdir^1/circles/^
    subdir^1/circles/^subdir^1/circles/^subdir^1/circles/^
    subdir^1/circles/^subdir^1/circles/^subdir^1/circles/^
    subdir^1/circles/^subdir^1/circles/^subdir^1/circles/^
    subdir^1/circles/^subdir^1/circles/^subdir^1/circles/^
    subdir^1/circles/^subdir^1/circles/^subdir^1/circles/^
    subdir^1/circles/^subdir^2/baz.txt
```

Queries, $0 \leq size \leq 5k$		result size		trav. nodes		col. nodes	
Q_1	/src/eyed3/utils/cli.py	1	(<0.1%)	71,595	(0.5%)	1	(<0.1%)
Q_2	//tests//	685,754	(6.8%)	6,344,106	(45.4%)	828,906	(5.9%)
Q_3	//tests/*	148,198	(1.5%)	7,024,730	(50.3%)	148,234	(1.1%)
Q_4	/*/include//	17,890	(0.2%)	4,196,184	(30.0%)	22,902	(0.2%)
Q_5	/src//nonexist	0	(0%)	280,099	(2.0%)	0	(0%)
Q_6	/src//	186,286	(1.8%)	3,338	(<0.1%)	276,772	(2.0%)
Q_7	/src/include//	12,137	(0.1%)	75,617	(0.5%)	16,803	(0.1%)
Q_8	/src/*	9,629	(0.1%)	138,897	(1.0%)	9,629	(<0.1%)

(a) CAS queries with size $\in [0, 5000]$

Queries, $0 \leq size \leq 100k$		result size		trav. nodes		col. nodes	
Q_1	/src/eyed3/utils/cli.py	0	(0%)	15,801	(0.1%)	0	(0%)
Q_2	//tests//	363,243	(3.6%)	2,687,233	(19.2%)	450,738	(3.2%)
Q_3	//tests/*	39,048	(0.4%)	3,098,920	(22.2%)	39,049	(0.3%)
Q_4	/*/include//	4,464	(<0.1%)	1,498,371	(10.7%)	5,786	(<0.1%)
Q_5	/src//nonexist	0	(0%)	60,484	(0.4%)	0	(0%)
Q_6	/src//	39,980	(0.4%)	1,758	(<0.1%)	58,726	(0.4%)
Q_7	/src/include//	3,067	(<0.1%)	16,589	(0.1%)	4,250	(<0.1%)
Q_8	/src/*	2,088	(<0.1%)	27,505	(0.2%)	2,088	(<0.1%)

(b) CAS queries with size $\in [0, 1000]$

Table 8.: CAS queries with the number of results, the number of traversed and collected nodes.

Bibliography

[1] Jdk-6962931 : move interned strings out of the perm gen. https://bugs.java.com/bugdatabase/view_bug.do?bug_id=6962931. [Online; accessed 5-July-2020].

[2] V. Leis, A. Kemper, and T. Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 38–49. IEEE, 2013.

[3] R. C. Merkle. A digital signature based on a conventional encryption function. In C. Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[4] A. Pietri, D. Spinellis, and S. Zacchiroli. The software heritage graph dataset: Public software development under one roof. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 138–142, 2019.

[5] popular-3k-python dataset. https://docs.softwareheritage.org/devel/swh-dataset/graph/dataset.html#popular-3k-python. [Online; accessed 21-June-2020].

[6] Software heritage archive. https://www.softwareheritage.org/. [Online; accessed 14-June-2020].

[7] Software heritage mission. https://www.softwareheritage.org/mission/. [Online; accessed 9-June-2020].

[8] Software heritage relational schema. https://docs.softwareheritage.org/devel/swh-dataset/graph/schema.html. [Online; accessed 07-July-2020].

[9] K. Wellenzohn, M. H. Böhlen, and S. Helmer. Dynamic interleaving of content and structure for robust indexing of semi-structured hierarchical data. *PVLDB*, 13(10):1641–1653, 2020.