1. Introduction

1.1. Background

Computer users interact with digital files on a daily basis. This includes storing files, organising files, and retrieving files. The act of managing files and retrieving information is fundamental and necessary for general computer usage and many computer-based knowledge work. With modern systems and increased computer usage, the number of digital files on a user's system has increased significantly [1].

This is related to the field of Personal Information Management (PIM). PIM is concerned with the management of digital files on the computer, and how people store, organise, and retrieve information to complete tasks. The increasing number of digital files on computer systems has made it more important than ever to investigate effective methods of managing digital files.

The directory tree structure is the de facto standard for data storage. Since its usage in early operating systems such as the OS/360 by IBM, most operating systems nowadays including Microsoft Windows, macOS and Linux distributions use directory trees as the data storage method. In such a system, filenames are used to uniquely identify files, and files can be placed within directories or folders.

The most common way of storing data on a computer is by using a directory tree structure. Organising files in a directory tree is analogous to sorting paper documents in real life - users can organise files by placing them into folders and further subfolders. While this is an intuitive way of organising files due to its similarities to organisation in real life, it is not an efficient way of organising and recalling files.

Storage of files using a directory tree is trivial, since users can store files anywhere they wish. The main challenge arises when the user has a large amount of files, it can become difficult to organise and retrieve them. Users have to spend time to search through folders and subfolders, looking for the file they want. This makes file management a time-consuming task.

1.2. Problem Statement

Current file management systems are not efficient when it comes to managing a large number of files. Trying to search for a particular file can be a very time-consuming process. The main reason for this is that these systems are based on a hierarchical structure which makes it difficult to search for files based on their content [2]. This problem is compounded when the users have to deal with a large number of files which can be difficult to organise consistently and structurally.

There are many methods available for users to search for files. This can include search functions, browsing through folders, or even looking through the recent documents list. However, these methods can be time-consuming and often give imprecise results that fail to find the file the user wants. This is a major problem for users who have to deal with large numbers of files.

Another major problem with current file management systems is that they are not flexible when it comes to organising files. This is because they are based on a hierarchical structure which can be inflexible and difficult to use. This can make it difficult for users to organise their files in a way that is suitable for them.

1.3. Aim

The project aims to investigate and develop a tag-based file manager. The system would allow users to assign tags to their files, and then search for those files based on those tags. The aim of this project is to develop a file management system which is more efficient than current systems, and which enables users to organise their files in a way that is suitable for them.

Compared to a hierarchical directory structure, a tag-based approach would allow users to organise their files in a more flexible and efficient way. Users would be able to assign arbitrary tags to their files then search for those files using those tags. This lets users avoid the time-consuming task of classifying files and browsing folders, for example deciding which single folder to place a file into [1] . This would also enable users to search for files based on their content, and would make it easier for users to manage their files.

Users would still be able to organise their files into folders if they so choose. However, the system would not impose any strict structure on users and would allow them to store their files in any way they want. This is in contrast to existing tagging software which often override any existing hierarchical file structure and do not allow any hierarchical organisation.

The system should provide a search function which would enable users to search for files based on their tags. This would make it easier for users to find the files they are looking for. The system should also be able to handle a large number of files, such that the software remains performant in tagging and searching for large numbers of files.

The system should also be easy to use, such that users can use it without any prior knowledge of tagging or file management. This is in contrast to existing tagging software which often have a steep learning curve, and require users to have some knowledge of tagging before they can be used effectively.

1.4. Objectives

The objectives of the project are as follows:

- To identify existing challenges related to PIM and file management.
- To compare and evaluate existing tagging software available to users.
- To understand the needs of users when performing file management tasks.
- To gain an understanding of concepts related to PIM.
- To investigate the challenges related to file management.
- To develop a system to allow users to manage and search files using tags.
- To implement a user interface that is easy to learn and efficient to use.
- To evaluate the effectiveness of the prototype system through user testing.
- To evaluate the system against existing software systems.
- To conclude on the effectiveness of the system and to come up with recommendations for further work.

1.5. Research Questions

The research questions for this project are as follows:

- How effective are existing solutions when it comes to managing a large number of files?
- What do users need from a file management system in order to manage their files effectively?
- Can a tag-based approach be used to improve the efficiency of file management?

2. Literature Review

In this chapter, we will review the existing literature related to file management, tagging, and personal information management. This will include a review of the challenges related to file management, an evaluation of existing solutions, and an investigation of the needs of users when performing file management tasks.

2.1. Personal Information Management

Personal Information Management (PIM) is a field of research which is concerned with the management of digital files on the computer. This includes how people store, organise, and retrieve information to complete tasks [3]. PIM is a relatively new field, and has only been studied extensively in the last few years. PIM is focused on how people organise, maintain and retrieve information, and on methods that can improve these tasks. This is not limited to digital files, but can also include paper documents for example.

One of the aims of PIM is for people to have the right information in the right place, in the right form and with enough completeness and quality to meet the current need [3]. However this is not the case for most people. Often the necessary information may not be found by the user, or the information may arrive at an unsuitable time such that it cannot be used.

[4] mentions the idea of "keeping found things found", in which people store information in multiple location and multiple applications. If this is performed inconsistently, the information people need is scattered widely, which makes it even more difficult to maintain and organise information. This is known as information fragmentation [3].

2.2. File Management

File management can be defined as the process of storing, retrieving and manipulating files on a computer system. This includes tasks such as creating, copying, moving and deleting files. File management is a fundamental task for users when using computers.

Most modern operating systems use directory trees to store data within files. A directory tree is a hierarchical structure in which each node represents either a file or folder. In this type of structure, filenames are used to uniquely identify each file within the system. Users can create directories or folders to group together related files.

While current methods for managing digital files are adequate for small workloads, they can become less effective with larger file collections. With larger collections, file hierarchies tend to have deeper structures [5]. This lead to file name duplication [6], and longer times spent to retrieve files [7]. Large hierarchies also require the user to make many navigational decisions with many subdirectories per directory [8].

There are many methods that can be used to help deal with large file collections. These include using automated organisational methods, and providing users with better tools for managing their files. Automated methods can be used to help users organise their files into a more manageable structure. This may include grouping files based on content or context using artificial intelligence algorithms.

2.3. Tagging

A tag is a label or keyword that can be attached to an item. One benefit of using tags to categorise items is that items can be easily found and classified. This can be useful for many tasks, such as information retrieval and content organisation.

One benefit of tagging over hierarchical systems is that it is more flexible. This is because tags can be arbitrarily named depending on the user's needs, and then assigned to multiple items at once. Items do not have to fit into predefined categories.

Tagging rose into popularity through its usage in many websites in Web 2.0 [9]. Tagging is now widely-used, and can be found in many social media websites and online services. One example of this is category tags in news websites and blogs.

Tagging has also been used to help users organise their files on the computer. There are some tools available that allow users to tag their files, and then search for them using those tags. This provides a

more flexible way of organising files compared to traditional hierarchical directory structures. It also makes it easier for users to find the files they are looking for since they can search using any combination of tags.

Tagging has many advantages over traditional methods for managing digital files. The most significant advantage is that it enables users to search for files based on their content, rather than just their filename or location. This makes it much easier for users to find the files they are looking for. Another advantage of tagging is that it does not require any specific structure or organisation, which may make it easier for users to manage their files.

There are also some disadvantages associated with tagging. One of the main disadvantages is that manual tagging can be a time-consuming process, particularly if the user has a large number of files. Another disadvantage is that automatic tagging systems are not always accurate, and can often assign incorrect tags to files.

2.4. Existing Programs Review

There are many software solutions for using tags for file management. Some of them are standalone applications, while some may be integrated into the operating system.

2.4.1. Windows' Native Tagging System

All Windows versions since Windows 7 has built-in tagging functionality. However this functionality is well-hidden and difficult to access. On files that support tags, a field for a list of tags is available on the "Properties > Details" popup panel.

Windows also provides a method for searching for files based on tags. On the search bar in Windows Explorer, the syntax `tag:<tagname>` can be used to search for files containing a specific tag.

The main advantage of this solution is that it is integrated into Windows, requiring no additional software. This makes it easy for users to get started with the system, and makes it easy to use since users are likely already familiar with File Explorer and Windows Search.

The main issue with Windows' native tagging system is that not all filetypes support tags. For example, tags cannot be applied to PNG image files and various audio formats including WAV and MP3. This is a significant limitation given that PNG, WAV, MP3 and other formats are commonly-used, users are likely to have many files of these types.

Another issue with Windows' tagging system is that users can only assign tags to each file at a time. It is not possible to assign a single tag to multiple files at the same time. With a large number of files, tagging each file will become a very time-consuming process.

One more issue is the speed of Windows' search functionality. Windows does not cache or index the tags contained in each file. This means that the system needs to scan each file for the specific tag the user requested when performing a search. The time required to search for files grows proportionally with the number of files the user has.

2.4.2. TagSpaces

TagSpaces is an open source application which runs on Windows, macOS and Linux.

It provides basic features such as assigning tags to files, and searching files based on tags. TagSpaces tags files by directly renaming the file with a prefix containing the tags to be assigned, wrapped with square brackets.

One benefit of this approach is that it is able to track file movement. Files that are moved using the file default program can still be found by TagSpaces since the tags are part of the filename. This also means that a file's tags are viewable using any file manager by simply viewing the filename.

One issue with TagSpaces is that renaming files can break links between applications and files. For files that act as dependencies for other application, this method cannot be used. For example, an audio file `cat.wav` may get renamed to `[animal recorded short purr] cat.wav`. Any third-party applications that are linked to the `cat.wav` file will now be unable to find the file and give an error.

Another issue with TagSpaces is that the filenames can get very long as users assign more tags to files. With a large number of tags on a file, the original filename will be prefixed by a large amount of text, making it difficult to view the original file name using file managers. For example, a file name like `[photo 2019 holiday uk london beach sunny clouds sea me dan jason burger beer sunglasses pub] IMG_7690.png` would easily get truncated in file managers, the user will not be able to view the full file name without resizing the entire window.

2.4.3. Tabbles

Tabbles is a desktop application that only runs on Windows systems. It comes in several variants: individual, cloud, and LAN. The different versions differ in how tags are shared between users on the same network or users connected to the cloud.

Tabbles tags files using a backing database, separate from the files being tagged. When the user moves a file to a different folder, a background service is used to detect such file movements and update the database to use the new path.

It provides many features. It allows grouping tags into a hierarchy, creating auto-tag rules, and tagging of entire folders for example.

One advantage of Tabbles is its auto-tagging feature. This feature allows for automatic assignment of tags to files based on specific file conditions. This helps the user save time when organising and managing files by automating the task of tag-assignment.

One issue with the software is that tags are not transferrable between computers. Tags are stored in a database that is associated with the system. This makes it very difficult to transfer tags if the user needs to upgrade their operating system and reinstall their system, or if the user wishes to upload their files to cloud backup.

From personal testing, another one of the issues is its slow performance. The application takes a long time to execute searches. When loading a large list of files, it slows down significantly as it scans each file to generate image previews.

2.5. Programming Tools and Frameworks

For the development of this project, a variety of tools and frameworks will be used. In this section, I will explain my decisions when it comes to deciding what toolset to use.

2.5.1. Rust

Rust is a programming language designed by Graydon Hoare. It is designed to be memory-safe while still being performant [10]. This is enforced by the Rust compiler, which checks memory-safety rules during compilation.

Rust is increasingly adopted by companies and projects such as Microsoft, Amazon, Cloudflare, and Facebook [11]. It has also been integrated into the Linux kernel [12].

Though I have years of experience in high-level languages such as Python, Ruby, Javascript, I have very limited experience with low-level languages like C++ and Rust. However, since Rust is being increasingly adopted by industry, as well as its memory-safety features, performance and industry support which make it a good choice for this project, it was an obvious choice for me when deciding

upon a programming language for development. I take this opportunity to learn Rust as it will also help with my career.

2.5.2. Tauri

Tauri is a Rust framework that allows developing desktop applications using web technologies like HTML, CSS, JavaScript for the frontend, while using Rust for the backend [13].

Rust is a relatively new language compared to other languages. It has many user interface libraries, however most of them are in the early stages of development and not yet fully-featured [14]. Among these, Tauri stands out for its support of many platforms and its polished feature set.

Tauri uses web technologies for implementing user interfaces. This allows developers to use widgets and components from frontend frameworks such as Vue and React - developers are not limited to a small subset of widgets implemented by the library author. Tauri can also take advantage of existing tools and libraries from the web development ecosystem - this provides a more complete solution.

The Rust backend can be used to access the file system and other desktop-specific features. This provides a more complete solution compared to using web technologies alone, for example building a browser-based file manager.

2.5.3. Vue

Vue is a JavaScript frontend framework developed by Evan You. It is designed to be flexible and easy-to-use, enabling the efficient development of web applications.

Vue combines aspects from modern frontend frameworks such as Angular.js and React. It generates a virtual DOM and renders this to the real DOM. It has many features such as two-way binding, reactivity, computed properties, single file components (SFC).

Vue has developed into one of the most popular frontend JavaScript frameworks. In 2022 it reached 3rd place in terms of popularity, below React and Angular and above Next.js and Svelte [15].

I chose Vue as the frontend framework because it has good documentation and many third-party libraries, and is a well-supported framework used by companies such as Facebook, Netflix, and Adobe.

I have some experience using Vue, but only in a limited capacity - I have experience with this framework through one of the university modules I am studying. I decided on this opportunity both improve my skills with the framework as well as developing new skills related to desktop application development.

2.5.4. SQLite

SQLite is a file-based relational database management system developed by D. Richard Hipp. It is designed to be lightweight, making it a good choice for small desktop applications since it does not require any external dependencies such as running an entire SQL server application in the background.

SQLite has basic features including transactions, triggers, views and foreign key support so that queries can be easily made using JOIN clauses. This gives tagging software access to many useful query functionalities without the overhead of running a separate process for hosting a database connection.

SQLite has a wide range of applications, including being used as an embedded database in Firefox and Android. It has good documentation, many client libraries and an active community of users.

I chose SQLite because I have plenty of experience using databases from my own personal use and through a part-time developer role. I would be able to implement a tagging system using SQLite and Rust without much difficulty.

3. Requirements Analysis

This chapter covers the requirements identified for my application. The following requirements were obtained through a review of existing tagging software and other file management systems, and through online sources such as blogs and discussion forums.

3.1. Functional Requirements

1. The system should allow users to tag files with arbitrary tags.
2. The system should provide a search function which would enable users to search for files based on their tags. This would make it easier for users to find the files they are looking for.
3. The system should preserve any existing directory tree structures, avoiding automatic renaming or moving of files.
4. The system should be able to handle a large number of files, such that the software remains performant in tagging and searching for large numbers of files.
5. The system should be easy to use, such that users can use it without any prior knowledge of tagging or file management.
6. The system should be able to tag any type of file, including but not limited to images, videos, text files, PDFs.
7. The system should be cross-platform and work on Windows, macOS and Linux systems.
8. The system should have a graphical user interface for tagging files and searching for files using tags.
9. The system should allow tags to be assigned to multiple files at once, rather than individually tagging each file one-by-one. This would reduce the amount of time needed for tag assignment, particularly for large numbers of files.

The following requirements are extensions to the project and not mandatory as base functions of the system.

8. The system should be able to handle transferring of tags between different devices or systems. This would allow users to preserve their tags when transferring across multiple systems, and would also allow files to be stored on cloud backup services such as Dropbox or Google Drive.
9. The system should provide tagging suggestions based on the content of the file being tagged. This would help users save time when tagging files by automating the task of tag-assignment
10. The system should display previews of each file, including thumbnails for image files and extension icons for unrecognised file types.

3.2. Non-functional Requirements

1. The system should be able to execute and complete searches within 1 seconds for large file collections up to 10000 files.
2. The system should have a response time of less than 500ms for all file operations such as opening, renaming and tagging files.
3. The application's memory usage should be below 50MB for small collections of up to 1000 files. For larger collections above 10000 files, the application's memory usage should not exceed 100MB.
4. The source code should be well-documented such that it is easy to understand and maintain.
5. All user input should be validated both on the frontend and backend to ensure data integrity and security.

6. The application's graphical user interface should be consistent across all platforms, providing a familiar experience for users regardless of their platform choice.

4. Design

This section discusses the components that make up the application and how they interact with each other. I will firstly go over each of the main software components, and then give an overview of how these components interact with each other. Following this, I will talk about my decision to use SQLite for storing tags, as well as discuss the user interface design.

4.1. Components

The following components make up the application:

- Rust backend for file operations, database access and cross-platform compatibility
- Vue frontend for the graphical user interface
- SQLite database for storing files and tags

The Rust backend is responsible for file operations, database access and cross-platform compatibility. The Vue frontend is responsible for the graphical user interface. The SQLite database stores tags assigned to files by users. These components are explained in more detail below.

4.1.1. Rust Backend

The Rust backend is responsible for file operations, database access and cross-platform compatibility. The backend is implemented using the Rust programming language and the Tauri framework.

The file operations module is responsible for listing and reading files in filesystem. Since the application writes to a database and not to files directly, this module is mainly focused on read operations.

The database access module is responsible for interacting with the SQLite database. It contains functions for opening a database connection, executing SQL queries, and closing a database connection. This module also implements helper functions for inserting tags into the database and retrieving tags from the database.

The cross-platform compatibility module is responsible for providing platform-specific functionality. This includes functions for opening an operating system specific file explorer, getting the path of common directories such as the home directory, and checking if a path is valid on the current platform.

4.1.2. Vue Frontend

The Vue frontend is responsible for the graphical user interface. It is implemented using the Vue JavaScript framework and TailwindCSS for styling.

The frontend consists of a single HTML file which loads the Vue JavaScript library and the TailwindCSS stylesheet.

The interface contains elements for displaying the application's toolbar, search and main content area. These elements are populated with data from the Rust backend using Tauri's command system.

The toolbar will contains buttons for opening repositories, which are directories with a centralised database that tracks tags with files. The toolbar will also contain operations for tagging files, and basic file operations such as renaming and deleting files. The main content area contains a list of items, which can be either files and directories, as well as previews and information about each file.

The frontend uses TailwindCSS for consistent styling across all HTML renderers on different platforms.

4.1.3. SQLite Database

The SQLite database stores tags assigned to files by users. The database is implemented using the SQLite 3 library and its full-text search extension. Please refer to [Database Schema] for the SQL schema.

The “items” table contains three columns - an autoincrementing ID, the path of the file, and a list of tags. The path column is used to store the relative path of the file to the root of the repository. The tag name column is used to store a list of tags assigned to the file.

The “items_fts” table is a virtual table using SQLite’s FTS5 full-text search extension. It allows quick indexing and querying of any text-based columns. In this case, the virtual table is used to index and search the tag name column on the “items” table.

The database is accessed by the Rust backend using the `rusqlite` Rust crate. Functions are provided for opening a database connection, executing SQL queries, and closing a database connection.

4.1.4. User Interface Design

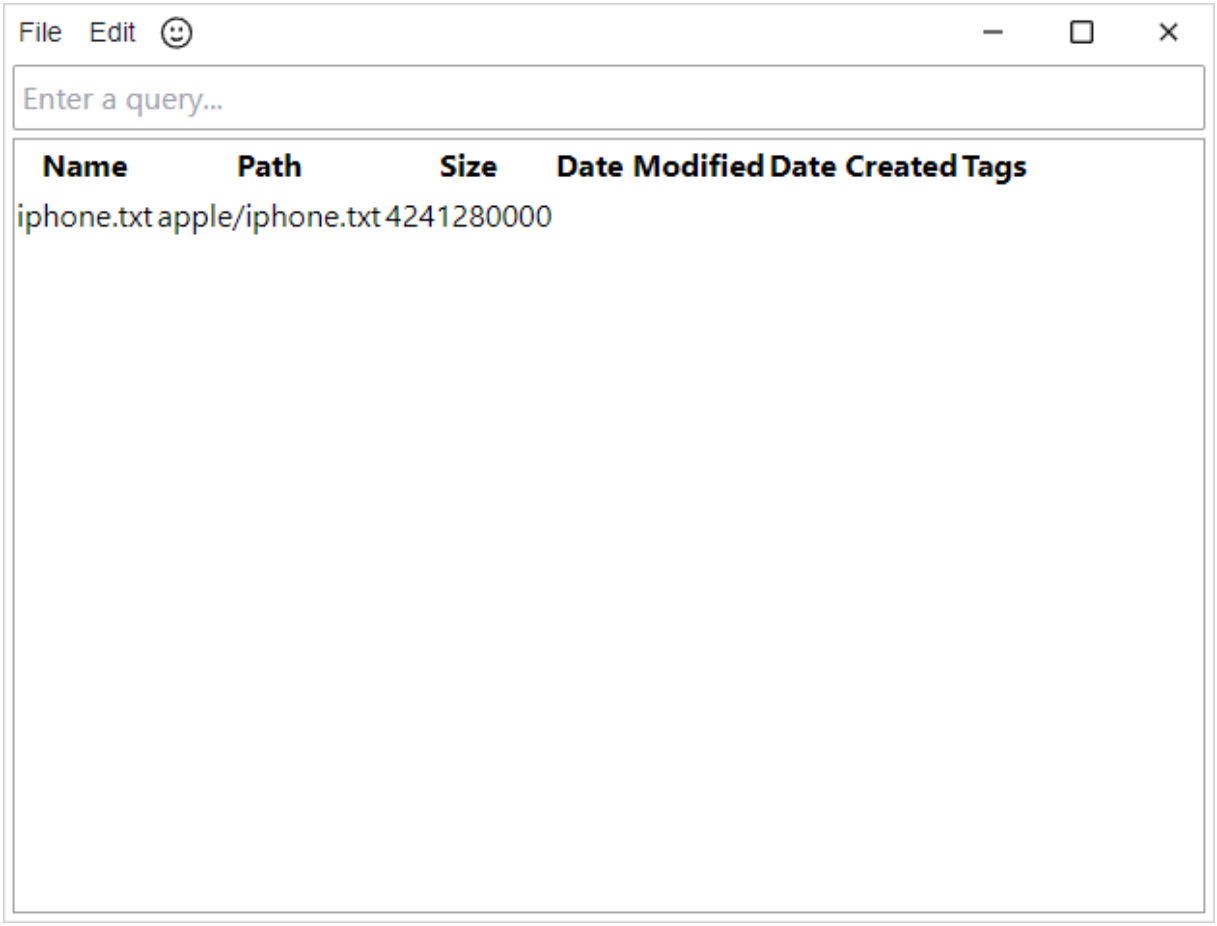

Figure 1: Work-in-progress UI, 26 November 2022.

The user interface is designed to be simple and easy-to-use. It consists of a toolbar, a search bar and the main content area. The toolbar contains buttons for opening files and directories, tagging files, renaming files and deleting files. The search bar allows the user to input arbitrary queries. The main content area contains a list of files and directories, as well as previews and information about each file.

The application is designed to be used with a mouse and keyboard. The toolbar buttons and main content list can be clicked using the mouse. Files and directories can also be opened by double-

clicking them with the mouse. Keyboard shortcuts are provided for focusing on the search bar (Ctrl+F) and tagging files (Ctrl+T).

When tagging files, the user is presented with a sidebar that displays information about the selected file in the main content area. In the sidebar users can edit the list of tags for the selected file. If the user has selected multiple files, the sidebar will show common tags between all selected files, and any new tags will be added to all selected files.

The user interface is designed to be consistent across all platforms. The application should look and feel the same on Windows, macOS and Linux systems.

4.2. Searching for Items

The application is designed to be a file manager. It should allow users to search for files using tags, but at the same time should not disregard the actual file structure of the underlying file tree. Therefore, users should be able to search for files not only using tags, but also using file attributes such as the file path.

The application uses SQLite's FTS5 full-text search extension to facilitate searching files by tag. When combined with other SQL conditions, the following is one example of such query:

```
SELECT i.id, i.path, i.tags, i.meta_tags
FROM items i
INNER JOIN
    tag_query tq ON tq.id = i.id
WHERE
    tag_query = '(a NOT b)'
    AND i.path LIKE 'samples/%';
```

The query searches for items that are under the path `samples/` and have the tag `"a"` but don't have the tag `"b"`.

To keep the software accessible and usable to as many users as possible, the software should not expect users to enter SQL queries directly since most computer users do not have experience in SQL. For more complex queries, the resulting SQL statement would become difficult to write and to understand.

As such, the application provides a plain-text query language that gets converted into SQL behind the scenes. The above query may be expressed in this new query language as follows:

```
a -b in:samples/
```

Terms such as `a` and `b` are treated as tag names, while terms with the prefix `in:` are file path queries. Terms that do not have any operators between them are implicitly joined with an `AND` group. Additional operators such as `|` and `-` allow grouping terms using boolean `OR` and `NOT` queries.

The application should convert the above plain-text query into the same SQL statement above when executing the query.

5. Implementation

5.1. Parsing Plain Text Queries

To support the plain-text query language described in [Searching for Items], the application needs to implement a compiler that translates from the plain-text query language to SQL.

This is one of the major challenges when implementing the software. In order to support arbitrarily-complicated queries, the compiler must be able to handle any combination of search operators and many edge cases.

Since the application fetches data from the same tables for every query, the `SELECT` and `JOIN` clauses of the SQL statement stay constant - I only need to consider the `WHERE` clause of the SQL statement when converting from a plain-text query. The following is the SQL template used in the compiler:

```
SELECT i.id, i.path, i.tags, i.meta_tags
FROM items i
INNER JOIN
    tag_query tq ON tq.id = i.id
WHERE
    :converted_plain_text_query
```

When given a plain-text query, the application converts it into a SQL `WHERE` clause, then inserts it into the template. However, due to differences between the plain-text and SQL languages, there is no straight-forward way to convert plain-text queries to SQL `WHERE` clauses. I will explain those differences by examining a few cases.

5.1.1. Query Edge Cases

5.1.1.1. Case 1: FTS5-only queries

```
-- The plain-text query:
-- "Item with tags a and b, or items with tag c without tag d"
a b | c -d

-- The SQL WHERE clause
WHERE tag_query = '(a AND b) OR (c NOT d)'
```

In the simplest case, the plain-text query only contains tag queries. In terms of SQL, the plain-text query only queries using the FTS5 extension without accessing any other columns. In this case, the resulting WHERE clause will only contain one expression - the FTS5 expression. Conversion from plain-text to FTS5 is relatively simple, except for an edge case:

```
-- The plain-text query:
-- "Item with tags a and b, or items without tag d"
a b | -d

-- The SQL WHERE clause
WHERE tag_query = '(a AND b) OR (meta_tags:all NOT d)'
```

The FTS5 extension treats `NOT` as a binary operator, not a unary operator. This means that it is impossible to search for the negation of a single term. For example, the following query cannot be represented under FTS5: *"items that don't have the tag 'old'"*.

To circumvent this issue, I added a new column to the `items(id, path, tags)` table called `meta_tags`. The `meta_tags` column contains the string `"all"` by default, so all items in the table will gain a new `all` tag. I can then specify `meta_tags:all` in the FTS5 query to search for *"all items that have the tag 'all' in the 'meta_tags' column"*. I finally use `meta_tags:all` as the first operand in the `NOT` operator to implement unary negation of a single term.

5.1.1.2. Case 2: FTS5 and SQL queries without OR operands

```
-- The plain-text query:
-- "Item in the folder 'my_folder' with tags a and b"
a b in:my_folder

-- The SQL WHERE clause
WHERE tag_query = 'a AND b' AND i.path LIKE 'my\_folder%' ESCAPE '\'
```

For clarity, “FTS terms/expressions” will refer to tag searches like “a b”, while “SQL terms/expressions” will refer to non-tag searches like “in:my_folder”.

In this case, the WHERE clause requires multiple expressions in addition to the FTS expression. For each SQL term in the query, the WHERE clause must include a separate SQL expression for the term. All the rules and edge cases from Section 5.1.1.1 apply here. This case is similar to case 1 and is simple to handle.

5.1.1.3. Case 3: FTS5 and SQL queries with OR operands

```
-- The plain-text query:
-- "Either i) item in the folder 'my_folder' with tags a and b, or
--         ii) item in the folder 'other_folder' without the tag d
a b in:my_folder | -d in:other_folder

-- Incorrect SQL clause
-- This will not behave as expected
WHERE (
  (tag_query = 'a AND b'
    AND i.path LIKE 'my\_folder%' ESCAPE '\')
  OR
  (tag_query = 'meta_tags:all NOT d'
    AND i.path LIKE 'other\_folder%' ESCAPE '\')
)

-- The working SQL WHERE clause
WHERE (
  (i.id IN (SELECT id FROM tag_query('a AND b'))
    AND i.path LIKE 'my\_folder%' ESCAPE '\')
  OR
  (i.id IN (SELECT id FROM tag_query('meta_tags:all NOT d'))
    AND i.path LIKE 'other\_folder%' ESCAPE '\')
)
```

When the plain-text query contains FTS5 terms and SQL terms joined together with at least one OR operator, the resulting SQL statement must make use of subqueries.

This is due to two limitations of SQLite’s FTS5 extension. First, it only supports at most one FTS5 expression in the WHERE clause. Attempting to query using multiple FTS expressions will result in zero rows being returned. Second, it does not handle OR groups that contain a FTS5 expression correctly. A query such as *“a b | in:my_folder”* will have rows missing from the output.

To overcome this, I replace all FTS expressions with a subquery that contains the required FTS expression. In effect, this make the statement behave as expected.

5.1.2. Compiler Implementation

The compiler is implemented using two files, `parser.rs` and `convert.rs`.

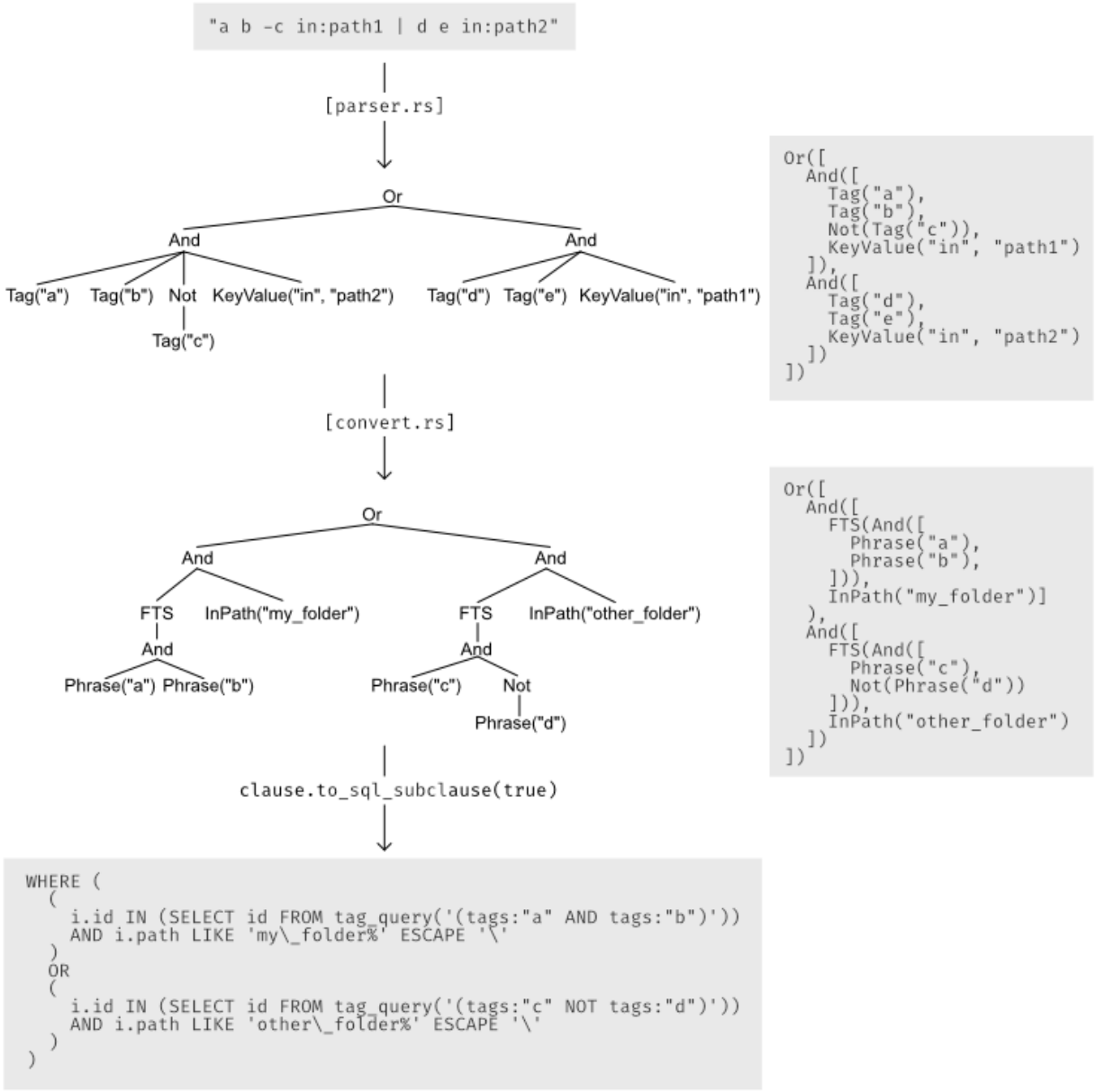

Figure 2: Diagram showing the process of converting a plain-text query to a SQL statement

The parser is implemented in `parser.rs` using the "nom" crate (library). Its purpose is to validate and parse a plain-text query into a parse tree.

The purpose of `convert.rs` is to convert the parse tree into an abstract syntax tree (AST). In this stage, it combines any FTS terms in the same level into a single `FTS()` object that represents a single FTS expression in the output SQL statement.

The conversion from the AST to SQL code is also handled by `convert.rs`. This is done by calling the `to_sql_clause(&self) -> String` method on the base of the tree, which then recursively calls `to_sql_subclause(&self, is_root: bool) -> String` on its child nodes.

Finally, the compiler inserts the SQL clause into the SQL template to obtain the final SQL statement. The statement can then be used to query items in the database.

The final number of lines of code in the compiler module was 1149 lines.

5.2. Testing

Testing of the application involved implementing unit tests for each application module, as well as integration tests for testing how the different modules integrate together.

At the moment, 104 unit and integration tests have been implemented. The test output is included in the appendix.

Unit tests for each module etc

Implemented.

5.3. Extension 1 - Directory watcher

The ability to track file movement is essential to this application. When implemented, it allows the application to preserve tags on a file when the user moves a file to a new location. However, implementing this on the Windows operating system presents a major challenge.

5.3.1. Detecting File Movement on Windows

Windows provides a native API to watch a directory for changes. However, one major issue with this API is that it is unable to detect file movement. The API can emit events for file creation, removal, and in-place renames. However, file movement from one directory to another is simply detected as a pair of file-create and file-remove events.

This prevents it from being able to be used directly as the watcher for the application. When used as the watcher for the application, the application is unable to preserve tags on a file when the user moves a file to a different folder. This is a major issue for many existing tag-based file managers as well - they often lack the ability to track file movement, have unorthodox solutions that affect user usability, or make it the user's responsibility to manually update tags after file movement.

To solve this issue, I implemented an asynchronous event handler that takes Windows' native events as input, and automatically resolves file movement by checking the file paths of the received events.

5.3.2. Structure

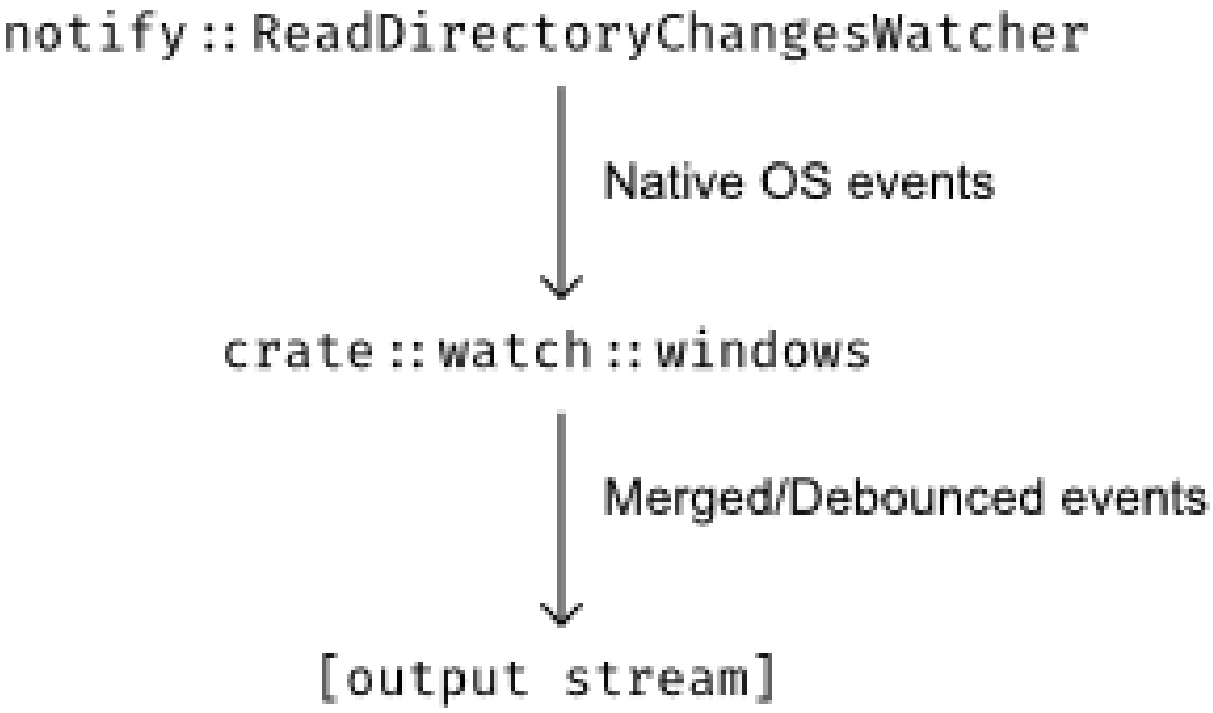

Figure 3: Diagram showing the flow of a Windows event from the watcher to the output

The final watcher makes heavy use of concurrent programming. The basic structure of the watcher is as follows:

- An instance of `ReadDirectoryChangesWatcher` from the `notify` crate, spawning native operating system events and sending them in a separate async task.
- An event handler that takes native OS events and processes them, finally sending them to the output receiver.

5.3.3. Event Handler Algorithm

The event handler receives new events in an infinite loop. It performs a different action depending on the type of event received.

When a delete event is received, it may correspond to either a file movement or a file deletion. The algorithm stores this event in a list, and adds to the event an expiry date. If the path is determined to be a rename event before the expiry time, then the algorithm sends a rename event. Otherwise, the path expires and is treated as a delete event, which the algorithm sends to the output.

When a create event is received, it may correspond to either a file movement or a file creation. The algorithm goes through the list of recently-deleted paths and checks if any path has the same file name as this create event. If a matching path is found, it marks the path as a rename event, then sends a rename event to the output. Otherwise, the algorithm returns the create event as-is.

If the list of recently-deleted paths is empty, the infinite loop blocks indefinitely while waiting for a new event from the `notify` watcher. If the list is not empty, the infinite loop will wait for the new event but timeout on the next earliest expiry time in the list. If a timeout occurs, it removes the associated path from the list and restarts the loop.

The full code for the event handler is included in the appendices at [Directory Watcher Event Handler].

5.4. Extension 2 - Searching based on file path

Allow searching for files using their path, so users can use the application just like any typical file manager.

Not yet implemented.

5.5. Extension X - Audio sample categorisation

Use deep learning to categorise audio files, then automatically tag them.

Not yet implemented.

5.6. Extension X - Query simplification

Simplify plain text query before converting to sql. Should use Disjunctive normal form: https://en.wikipedia.org/wiki/Disjunctive_normal_form

Not yet implemented.

6. Risk assessment

Risk|Impact|Likelihood|Rating|Preventative actions –|–|-|-|— Cannot finish initial implementation by March|Cannot complete draft report with sufficient information|Low|High|Break down project into basic functionality and extensions, focus solely on basic functionality before moving onto extensions Fail to find any users for testing|Cannot complete project evaluation|Low|High|Start finding users for testing as soon as initial implementation is complete

7. Appendices

7.1. Database Schema

```sql
CREATE TABLE items (
  id INTEGER PRIMARY KEY,
  path TEXT UNIQUE NOT NULL,
  tags TEXT NOT NULL,
  meta_tags TEXT NOT NULL DEFAULT 'all'
);

-- FTS5 Documentation:
-- https://www.sqlite.org/fts5.html
CREATE VIRTUAL TABLE tag_query USING fts5 (
  -- Include columns to be stored on this virtual table:
  -- Include the `id` column so I can join it to `items`, but don't index with FTS
  id UNINDEXED,
  -- Include the `tags` column to index them
  tags,
  -- a 'meta' column that stores additional tags, e.g. 'all'
  meta_tags,

  -- Make this an external content table (don't store the data in this table, but
reference
  -- the original table)
  content=items,
  content_rowid=id,

  -- Use the Unicode61 tokenizer
  -- https://www.sqlite.org/fts5.html#unicode61_tokenizer
  tokenize="unicode61"
);

CREATE TRIGGER items_trigger_ai AFTER INSERT ON items BEGIN
  INSERT INTO tag_query(id, tags, meta_tags) VALUES (NEW.id, NEW.tags,
NEW.meta_tags);
END;

CREATE TRIGGER items_trigger_ad AFTER DELETE ON items BEGIN
  INSERT INTO tag_query(tag_query, id, tags, meta_tags) VALUES('delete', OLD.id,
OLD.tags, OLD.meta_tags);
END;

CREATE TRIGGER items_trigger_au AFTER UPDATE ON items BEGIN
  INSERT INTO tag_query(tag_query, id, tags, meta_tags) VALUES('delete', OLD.id,
old.tags, old.meta_tags);
  INSERT INTO tag_query(id, tags, meta_tags) VALUES (NEW.id, NEW.tags,
NEW.meta_tags);
END;
```

7.2. Directory Watcher Event Handler

```rust
fn clear_expired_records(
    recent_deleted_paths: &mut Vec<(Instant, PathBuf, EventAttributes)>,
    output_tx: &UnboundedSender<notify::Result<Event>>,
) {
    let now = Instant::now();
    recent_deleted_paths.retain(|(expires_at, path, attrs)| {
```

```rust
        if expires_at <= &now {
            let evt = Event {
                kind: Remove(RemoveKind::Any),
                paths: vec![path],
                attrs,
            };
            output_tx.send(Ok(evt)).unwrap();
            true
        } else {
            false
        }
    });
}

async fn event_handler(
    mut watcher_rx: UnboundedReceiver<notify::Result<Event>>,
    output_tx: UnboundedSender<notify::Result<Event>>,
) {
    let mut last_rename_from: Option<PathBuf> = None;
    let mut recent_deleted_paths: Vec<(Instant, PathBuf, EventAttributes)> = vec![];
    let mut res;
    loop {
        // If we have paths in the database, timeout until the next path's instant
        if recent_deleted_paths.len() > 0 {
            let next_wake_time = recent_deleted_paths.get(0).unwrap().0;
            match timeout_at(next_wake_time.clone(), watcher_rx.recv()).await {
                Ok(x) => {
                    // Didn't timeout, assign the return value to res
                    res = x;
                }
                Err(_) => {
                    // Timeout occurred, clear expired records from database and wait
again
                    clear_expired_records(&mut recent_deleted_paths, &output_tx);
                    continue;
                }
            }
        } else {
            // No paths in database, just wait for next record indefinitely
            res = watcher_rx.recv().await;
        }
        match res {
            Some(evt) => {
                if evt.is_err() {
                    output_tx.send(evt).unwrap();
                    continue;
                }
                let evt = evt.unwrap();
                match evt {
                    Event {
                        kind: Modify(Name(RenameMode::From)), mut paths, ..
                    } => {
                        if let Some(_) = last_rename_from {
                            panic!("Got multiple 'Rename From' events in a row!")
                        }
                        let path = paths.pop().unwrap();
```

```
                        last_rename_from = Some(path);
                        continue;
                    }
                    Event { kind: Modify(Name(RenameMode::To)), mut paths, .. } => {
                        let from_path = last_rename_from.take().expect(
                        "Got 'Rename To' event, but no 'Rename From' event happened
before this!",
                    );
                        let to_path = paths.pop().unwrap();
                        let evt = Event {
                            kind: Modify(Name(RenameMode::Both)),
                            paths: vec![from_path, to_path],
                            attrs: evt.attrs.clone(),
                        };
                        output_tx.send(Ok(evt)).unwrap();
                    }
                    Event { kind: Remove(RemoveKind::Any), mut paths, attrs } => {
                        assert_eq!(
                            paths.len(),
                            1,
                            "Number of created paths is not 1: {}",
                            paths.len()
                        );
                        let removed_path = paths.pop().unwrap();
                        let expires_at = Instant::now() + Duration::from_millis(10);
                        recent_deleted_paths.push((expires_at, removed_path, attrs));
                    }
                    Event { kind: Create(CreateKind::Any), paths, attrs } => {
                        assert_eq!(
                            paths.len(),
                            1,
                            "Number of created paths is not 1: {}",
                            paths.len()
                        );
                        let created_path = paths.get(0).unwrap().clone();
                        let mut deleted_path_match_id: Option<usize> = None;
                        for i in 0..recent_deleted_paths.len() {
                            let deleted_path =
&recent_deleted_paths.get(i).unwrap().1;
                            let created_name = created_path
                                .file_name()
                                .expect("Path doesn't have file name");
                            let deleted_name = deleted_path
                                .file_name()
                                .expect("Path doesn't have file name");
                            if created_name == deleted_name {
                                deleted_path_match_id = Some(i);
                                break;
                            }
                        }
                        match deleted_path_match_id {
                            Some(i) => {
                                let deleted_path_match =
recent_deleted_paths.remove(i).1;
                                let evt = Event {
                                    kind: Modify(Name(RenameMode::Both)),
```

```
                                    paths: vec![deleted_path_match,
created_path.to_path_buf()],
                                    attrs,
                                };
                                output_tx.send(Ok(evt)).unwrap();
                            }
                            None => {
                                let evt = Event {
                                    kind: Create(CreateKind::Any),
                                    paths: vec![created_path],
                                    attrs,
                                };
                                output_tx.send(Ok(evt)).unwrap();
                            }
                        }
                    }
                    _ => output_tx.send(Ok(evt)).unwrap(),
                }
            }
            None => {
                // send remaining deleted paths to output
                for (_, path, attrs) in recent_deleted_paths {
                    let evt = Event {
                        kind: Remove(RemoveKind::Any),
                        paths: vec![path],
                        attrs,
                    };
                    output_tx.send(Ok(evt)).unwrap();
                }
                break;
            }
        }
    }
}
```

7.3. Test Output

```
test helpers::sql::test_fts5::no_quotes ... ok
test helpers::sql::test_fts5::both_quotes ... ok
test helpers::sql::test_fts5::single_quotes ... ok
test helpers::sql::test_fts5::double_quotes ... ok
test helpers::sql::test_like::both_quotes ... ok
test helpers::sql::test_like::escape_char_1 ... ok
test helpers::sql::test_like::single_quotes ... ok
test helpers::sql::test_like::double_quotes ... ok
test helpers::sql::test_like::no_quotes ... ok
test helpers::sql::test_like::percent_2 ... ok
test helpers::sql::test_like::escape_char_2 ... ok
test helpers::sql::test_like::percent_1 ... ok
test helpers::sql::test_like::underscore_2 ... ok
test query::convert::test_clauses::fts_3 ... ok
test query::convert::test_clauses::common_1 ... ok
test query::convert::test_clauses::common_2 ... ok
test query::convert::test_clauses::common_3 ... ok
test query::convert::test_clauses::fts_1 ... ok
test query::convert::test_clauses::fts_2 ... ok
test helpers::sql::test_like::underscore_1 ... ok
```

```
test query::convert::test_clauses::inpath_1 ... ok
test query::convert::test_clauses::inpath_3 ... ok
test query::convert::test_clauses::fts_4 ... ok
test query::convert::test_clauses::fts_5 ... ok
test query::convert::test_clauses::inpath_4 ... ok
test query::convert::test_clauses::inpath_2 ... ok
test query::convert::test_clauses::inpath_5 ... ok
test query::convert::test_clauses::inpath_6 ... ok
test query::convert::test_fts_query::and_1 ... ok
test query::convert::test_fts_query::and_2 ... ok
test query::convert::test_fts_query::neg_1 ... ok
test query::convert::test_fts_query::complex_1 ... ok
test query::convert::test_fts_query::neg_3 ... ok
test query::convert::test_fts_query::neg_2 ... ok
test query::convert::test_fts_query::neg_4 ... ok
test query::convert::test_fts_query::neg_5 ... ok
test query::convert::test_fts_query::or_1 ... ok
test query::convert::test_fts_query::or_2 ... ok
test query::convert::test_to_sql::fts_1 ... ok
test query::convert::test_to_sql::common_1 ... ok
test query::convert::test_to_sql::fts_2 ... ok
test query::convert::test_to_sql::inpath_1 ... ok
test query::convert::test_to_sql::fts_3 ... ok
test query::convert::test_to_sql::inpath_2 ... ok
test query::convert::test_to_sql::inpath_3 ... ok
test query::convert::test_to_sql::inpath_4 ... ok
test query::convert::test_to_sql::inpath_5 ... ok
test query::parser::expr_tests::and_or_1 ... ok
test query::parser::expr_tests::and_or_2 ... ok
test query::parser::expr_tests::common_1 ... ok
test query::parser::expr_tests::just_and_1 ... ok
test query::parser::expr_tests::just_and_2 ... ok
test query::parser::expr_tests::just_and_3 ... ok
test query::parser::expr_tests::complex_1 ... ok
test query::parser::expr_tests::just_and_4 ... ok
test query::parser::expr_tests::just_or_2 ... ok
test query::parser::expr_tests::just_or_1 ... ok
test query::parser::expr_tests::just_or_3 ... ok
test query::parser::expr_tests::just_or_4 ... ok
test query::parser::expr_tests::parens_1 ... ok
test query::parser::expr_tests::not_1 ... ok
test query::parser::expr_tests::not_2 ... ok
test query::parser::expr_tests::parens_2 ... ok
test query::parser::expr_tests::parens_3 ... ok
test query::parser::expr_tests::parens_4 ... ok
test query::parser::expr_tests::string_tags_1 ... ok
test query::parser::expr_tests::string_tags_2 ... ok
test query::parser::expr_tests::string_tags_3 ... ok
test query::parser::tests::test_literal ... ok
test query::parser::tests::test_key_value ... ok
test query::parser::tests::test_string ... ok
test query::parser::tests::test_tag ... ok
test query::tests::common_1 ... ok
test query::tests::common_2 ... ok
test query::tests::empty ... ok
test query::tests::common_3 ... ok
```

```
test watch::tests::basic_test ... ok
test scan::tests::scans_files_in_folder ... ok
test watch::tests::file_creations_01 ... ok
test watch::tests::file_removes_01 ... ok
test watch::tests::file_creations_02 ... ok
test watch::tests::file_removes_02 ... ok
test watch::tests::file_renames_01 ... ok
test watch::tests::file_renames_02 ... ok
test watch::tests::file_renames_03 ... ok
test repo::tests::check_tables_of_newly_created_database ... ok
test repo::tests::can_get_item_by_id ... ok
test repo::tests::can_remove_item_by_path ... ok
test tests::query_repo::query_1 ... ok
test tests::query_repo::query_4 ... ok
test repo::tests::cant_insert_duplicate_items ... ok
test tests::query_repo::query_2 ... ok
test repo::tests::can_insert_items ... ok
test tests::query_repo::query_3 ... ok
test repo::tests::can_get_item_by_path ... ok
test repo::tests::can_get_all_items ... ok
test repo::tests::can_query_items ... ok
test repo::tests::can_update_item_path ... ok
test repo::tests::can_remove_item_by_id ... ok
test repo::tests::can_update_item_tags ... ok
test repo::tests::query_test ... ok
test repo::tests::query_test_2 ... ok
test scan::tests::benchmark ... ok
test repo::tests::scan_integration::my_test ... ok
```

8. Bibliography

Bibliography

[1] J. D. Dinneen, and C.-A. Julien, “The ubiquitous digital file: a review of file management research,” *J. Assoc. Inf. Sci. Technol.*, vol. 71, no. 1, pp. 1–32, 2020.

[10] A. Balasubramanian, M. S. Baranowski, et al., “System programming in rust: beyond safety,” in *Proc. 16th Workshop Hot Topics Operating Syst.*, 2017, pp. 156–161.

[11] L. Davison, “Adoption of rust: who's using it and how,” The New Stack, 2022. Accessed: Nov. 21, 2022. [Online]. Available: https://thenewstack.io/adoption-of-rust-whos-using-it-and-how/

[12] L. Proven, “Rust is coming to the linux kernel,” The Register, 2022. Accessed: Nov. 21, 2022. [Online]. Available: https://www.theregister.com/2022/09/16/rust_in_the_linux_kernel/

[13] Tauri Contributors, “Tauri-apps/tauri,” GitHub, 2022. Accessed: Nov. 21, 2022. [Online]. Available: https://github.com/tauri-apps/tauri

[14] shivshank, and dxenonb, “Are we gui yet?,” Are we GUI yet?, 2022. Accessed: Nov. 21, 2022. [Online]. Available: https://www.areweguiyet.com/

[15] Stack Overflow, “Stack overflow developer survey 2022,” Stack Overflow, 2022. Accessed: Nov. 28, 2022. [Online]. Available: https://survey.stackoverflow.co/2022/

[2] L. De Vocht, A. Denger, B. Stadlhofer, and K. Voit, “Formal experiment report: tagging files vs. placing files in a hierarchy,” Graz University of Technology, Graz, Austria, Feb. 2012. [Online].

Available: https://github.com/novoid/2011-01-tagstore-formal-experiment/blob/master/analysis_and_derived_data/Results_Report.pdf

[3] W. P. Jones, and J. Teevan, *Personal Information Management*, vol. 14, University of Washington Press Seattle, 2007.

[4] W. Jones, *Keeping Found Things Found: The Study and Practice of Personal Information Management*, Morgan Kaufmann, 2010.

[5] S. Henderson, and A. Srinivasan, "An empirical analysis of personal digital document structures," in *Symp. Human Interface*, 2009, pp. 394–403.

[6] S. Henderson, "Document duplication: how users (struggle to) manage file copies and versions," *Proc. Amer. Soc. Inf. Sci. Technol.*, vol. 48, no. 1, pp. 1–10, 2011.

[7] O. Bergman, S. Whittaker, M. Sanderson, R. Nachmias, and A. Ramamoorthy, "The effect of folder structure on personal file navigation," *J. Amer. Soc. Inf. Sci. Technol.*, vol. 61, no. 12, pp. 2426–2441, 2010.

[8] B. J. H. and
Andy Dong and
R. Palmer and Hamish C. McAlpine, "Organizing and managing personal electronic files: A mechanical
engineer's perspective," *ACM Trans. Inf. Syst.*, vol. 26, no. 4, pp. 1–40, 2008, doi: 10.1145/1402256.1402262. [Online]. Available: https://doi.org/10.1145/1402256.1402262

[9] G. Smith, *Tagging: People-Powered Metadata for the Social Web*, New Riders, 2007.