Vol. 8, No. 1, February 2026

Journal of Information Systems and Informatics

Published By Asosiasi Doktor Sistem Informasi Indonesia

Design and Implementation of a Hierarchically Interoperable Tag-Based File System using FUSE (PreTFS)

Lie Steven Staria Nugraha[1], Fahri Firdausillah[2]

[1,2]Department of Computer Science, Universitas Dian Nuswantoro, Semarang, Indonesia

Received:
November 19, 2025
Revised:
January 14, 2026
Accepted:
January 29, 2026
Published:
February 19, 2026

Corresponding Author:
Author Name*:
Lie Steven Staria Nugraha
Email*:
nugrahasteven@gmail.com

DOI:
10.63158/journalisi.v8i1.1416

Abstract. Traditional hierarchical file systems make semantic organization awkward: a file that naturally belongs to multiple contexts must be forced into a single directory, leaving users to choose an arbitrary location or rely on duplication, linking, or search. This paper presents the design, prototype, and evaluation of a file system that preserves conventional hierarchical standards while adding an opt-in, tag-based semantic layer for multi-context categorization. We describe (i) a design in which tags are represented as directories with reserved, prefixed names and tag intersections are expressed through ordinary path nesting, and (ii) a proof-of-concept implementation that validates feasibility in practice. The implementation, PreTFS, is built as a FUSE (Filesystem in User Space) file system and uses SQLite to store file metadata and content. Results show that the design is realizable and remains compatible with conventional applications and workflows without external tools or specialized APIs. Benchmarking against a native kernel file system (btrfs) reveals expected overheads from user-space indirection and metadata management, measuring approximately ~2–73 ms for metadata-oriented operations and ~1–160 ms for file-content operations. These costs indicate the approach is practical for small-scale environments such as personal information management, where semantic flexibility and interoperability can outweigh peak performance. The novelty lies in a simple, hierarchically interoperable tagging design that enables semantic categorization through standard directory navigation.

Keywords: Hierarchical Interoperability; Tag-Based Semantic File System; FUSE; Personal Information Management; Metadata Indexing

1. INTRODUCTION

Modern computing relies on file systems to store data persistently and to expose operating-system services for creating, reading, updating, and managing that data [1]. Recent research has substantially advanced storage performance [2], [3] and system stability [4], yet the user-facing interface for organizing and retrieving files remains largely unchanged. In practice, file systems are still "wed" to hierarchical directory trees whose core design dates back to the 1970s [5], [6]. Standards and conventions such as the Filesystem Hierarchy Standard (FHS) on Linux, as well as the analogous conventions on macOS and Windows, reinforce this structure as the default. This is not merely an engineering artifact: a recent survey shows that navigating hierarchical directories to retrieve information is the most frequent practice among personal information management activities [7], underscoring how deeply ingrained the hierarchy has become—even as the scale and complexity of stored information continues to grow.

The central problem with hierarchical file systems is that they cannot naturally represent overlapping semantic membership. For a file f and two semantic directories A and B, hierarchical structures struggle to express $f \in A \cap B$without forcing a trade-off. A file like *biochemistry.md* spans two semantic groupings—*biology* and *chemistry*—but the directory tree forces it to "live" in only one location. Common workarounds such as duplication, hard links, or reliance on expensive searches each impose tangible costs: duplication risks divergence, linking adds management complexity and can behave inconsistently across tools, and search shifts the burden to retrieval time and often degrades with depth and scale [1], [8], [9], [10]. These limitations are not purely theoretical; they conflict with how users often prefer to organize information. For instance, a survey of 74 postgraduate students found that most prefer an unconstrained approach to personal information management rather than one constrained by time, activity, or topic [11], whereas hierarchical organization implicitly forces such constraints by demanding a single "correct" place for each item.

Semantic file systems are a natural candidate for addressing this mismatch because they index and organize files by meaning rather than by location alone [12]. In particular, tag- or category-based semantic file systems associate multiple identifiers with a file, allowing

the same file to appear in multiple semantic categories without duplication. However, the key barrier to adoption is not the concept of tagging itself but compatibility with the surrounding ecosystem. Most software—from command-line utilities like ls, tree, and pwd to graphical file explorers—assumes that directories are concrete hierarchical containers rather than query-defined views. As a result, any semantic alternative that disrupts the standard hierarchy or requires specialized interfaces risks being impractical for everyday use. We therefore define a fully hierarchically interoperable file system as one that (1) leaves the existing hierarchy untouched unless explicitly instructed, and (2) works with standard OS tools without extra utilities or APIs.

Existing semantic file systems fall short of this interoperability goal in different ways. Early systems, such as the original Semantic File Systems work [13] and TagFS [14], represent tags through virtual directories, but the former can hide or distort expected behavior in path-based tools such as pwd, while the latter treats every directory as a virtual directory and thereby violates the requirement that the original hierarchy remain intact unless explicitly changed. More recent conceptual models, including Linked Tree Tags [15] and its AttFS extension [16], enrich tagging with boolean operators and attributes, but they rely on cumbersome operator syntax (e.g., Λ, V, ¬) that is not available on standard keyboard layouts and, importantly, remain largely unvalidated in practice due to a lack of implementations. Other systems such as 360° SFS [8] attempt to reduce manual effort by automating tag suggestions, yet they encode tags and intersections through filename postfixing, which increases cognitive overhead and undermines the familiar "directories as navigable places" interaction model. A different line of work embeds semantic retrieval inside dedicated applications or custom query interfaces [1], [17], but this breaks interoperability by moving core file navigation outside the standard OS toolchain.

This paper addresses the resulting gap: prior work either improves semantic expressiveness at the expense of hierarchical interoperability, or preserves conventional navigation while failing to support intuitive, practical representations of semantic overlap. We propose a hierarchically interoperable, tag-based semantic file system in which tags are represented as directories with prefixed names, and intersections of tags are represented as nested prefixed directories, enabling semantic composition through

ordinary path navigation rather than special syntax. In doing so, we contribute both a concrete design (Section 2.1) and a working FUSE implementation (Section 2.2) that can be used with standard tools without requiring external utilities or APIs. Finally, we empirically evaluate the performance of the implementation relative to a native kernel file system (btrfs) (Section 3.2), quantifying the practical trade-offs of adding semantic views while maintaining hierarchical interoperability.

2. METHODS

To outline the research procedure done for this paper, Figure 1 visualizes the research process, whilst Table 1 describes each process in detail, along with their respective result or findings. Together, these visualizations anchor our procedural narrative, linking each methodological step to the ensuing implementation choices and empirical evaluations. The figure highlights the iterative back-and-forth between design decisions and validation, while the table enumerates the concrete activities and observed outcomes for every numbered phase of the study.

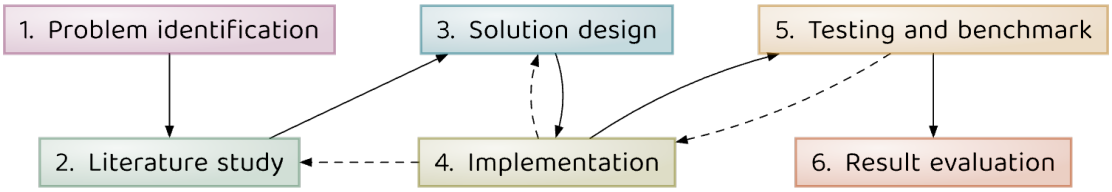

Figure 1. Step-by-step diagram representing the research methodology. Dashed lines represent backtracking/revision.

Table 1. Sequential detail on how this research is conducted

Step	Description	Summary of results or findings
1. Problem identification	Analyze the current file system design architecture landscape. Generalize the problem.	The findings of this step have been thoroughly discussed within Section 1 of this paper. In brief, hierarchical structures cannot represent semantic categories effectively.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

Step	Description	Summary of results or findings
2. Literature study	Ensure no existing solutions. Exemplify existing file systems. Consult software documentations.	Also discussed within Section 1, no existing solution solves the problem effectively.
3. Solution design	Hypothesize a unique design that effectively solves the problem. Keep implementation feasibility in mind (hierarchic interoperability).	Will be discussed in Section 2.1. Essentially, use prefixes to represent semantic directories and nested prefixed directories to represent semantic intersections.
4. Implementation	Implement the design. Revise the design if needed. Review more literature and documentation for clarity.	The design is implemented using FUSE, Rust and SQLite, as a file system called PreTFS. Implementation details will be described in Section 2.2.
5. Testing and benchmark	Ensure the implementation works. Fix or improve the implementation accordingly. Analyze the final performance.	The implementation works. Performance comparison with btrfs is described in Section 3.2.
6. Result evaluation	Draw conclusions from the overall research. Ensure that the problem is solved. Identify the limitations of the file system. State ideas on what can be done in future works.	The design is feasible, as proven by the implementation. The proposed file system is fit for metadata-rich use cases, such as personal information management environments. Limitations of the design are described in Section 3.3.1. Ideas for further research are discussed afterwards.

2.1. Design

This paper proposes a tag-based file system design that fulfills the hierarchical interoperability qualifications described in Section 1. The proposed file system represents directories with prefixed names as semantic categories, and nested prefixed directories as category intersections. Figure 2 illustrates an example structure for the proposed design. Its nodes represent files and directories, whilst their edges represent their hierarchical relationship.

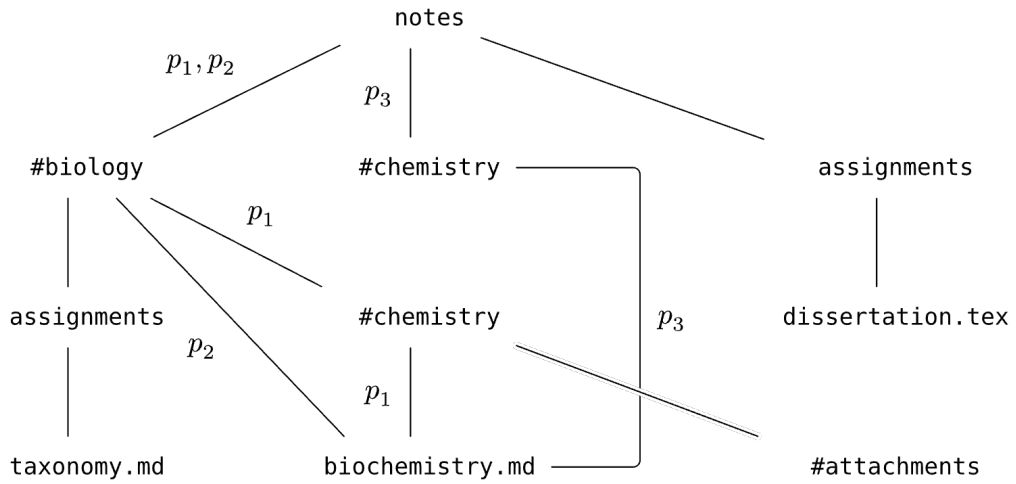

Figure 2. Top-down tree-like file system structure, for the proposed file system design's note taking example

In Figure 2, each node can be reached by one or many paths. For example, labels p_1, p_2, and p_3 represent different paths leading to the same biochemistry.md file. Table 2 shows what each p_i represents, along with the reason why each path may be used to access biochemistry.md.

Table 2. Mapping between each p_i path for biochemistry.md and its path reasoning

p_i	Path representation	Path reasoning
p_1	/notes/#biology/#chemistry/biochemistry.md	It is associated with the chemistry tag
p_2	/notes/#biology/biochemistry.md	It is associated with the biology tag
p_3	/notes/#biology/#chemistry/biochemistry.md	It is associated with the biology and chemistry tag

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

On the other hand, there are two assignments directory: one under notes, and another under /notes/#biology. Neither of these directories is prefixed. However, if they were prefixed directories, taxonomy.md would also appear in /notes/assignments, since it would be associated with the assignments tag. While the file system design allows for user-defined delimiters, the hash symbol serves as the default prefix. This selection is driven by two primary factors. First, the symbol offers semantic familiarity, leveraging the ubiquitous "hashtag" convention found in social media to denote categories. Second, it avoids namespace collisions with reserved system conventions, most notably the dot prefix used to omit hidden files. However, the hash prefix does have one notable drawback, which is that in most shells (e.g., zsh, bash, fish), the hash prefix is used to specify comments. This drawback can be circumvented by *escaping* or wrapping the hash prefix in a string. For a user to interact with the proposed file system, some operations must be facilitated by the file system. Aside from the standard file system operations, such as reading or writing bytes into a file, Table 3 indexes a non-comprehensive list of relevant operations that the user of the proposed file system may do.

Table 3. Summary of relevant operations to be executed by the user

Operation	Summary
Create new tag	Create a directory with a unique prefixed name.
Create file tag association	Place a file inside a prefixed directory.
Update file tag associations	Move files from one prefixed directory to another.
Index tagged files	Read the contents of a prefixed directory.1
Delete tags	Remove every associated file.2

[1] Prefixed directories are entirely structured hierarchically. To tie back to the previous example in Figure 2, the /notes/#biology/#chemistry/#attachments prefixed directory does not appear in /notes/#chemistry. This wouldn't be the case if the #attachments directory were an unprefixed directory.

[2] Removing a prefixed directory requires all its prefixed directory children to be deleted recursively. For example, if the user wants to delete the /notes/#biology directory in Figure 2, /notes/#biology/#chemistry must be deleted first, unlike /notes/#biology/biochemistry.md, which doesn't have to be deleted first.

2.2. Implementation

Traditionally, within UNIX operating systems, file systems are implemented within the kernel space. To allow multiple file systems to be mounted simultaneously, UNIX operating systems like Linux implement a *Virtual File System* (often abbreviated as VFS). In brief, the role of a VFS is to provide a common interface between file-related function calls and an underlying file system. Each function call is mapped to a corresponding file system function, depending on which file system is operated on. Some of the more common kernel space file systems supported by the VFS are:

1) *ext4* (fourth extended filesystem), the most recent version of the most used file system in Linux
2) *NTFS* (New Technology File System), used in Windows operating systems, is now interoperable with Linux
3) *exFAT* (Extended File Allocation Table), mostly used in portable storage devices such as SD cards and USB flash drives

While the VFS layer provides mounting flexibility, it traditionally places a deployment burden on the user, often necessitating custom kernel modules or manual patching to support new file systems. Filesystem in Userspace (FUSE) circumvents this complexity by relying on a single, ubiquitous kernel module. Instead of requiring a bespoke kernel driver, a FUSE-based file system communicates with the kernel via a user-space interface. This architecture significantly lowers the barrier for implementing complex features, such as network integration via ssh (e.g., sshfs) or interaction with relational databases (e.g., sqlitefs), which would be prohibitively difficult to develop directly within the kernel.

As proof of concept, a FUSE implementation of the proposed design (presented in Section 2.1), called PreTFS3 (Prefix-Tag File System), will be presented within this section. Rust was selected as the programming language for this implementation due to its compile-time safety, modern design, and low-level FUSE support. To facilitate this implementation, an SQLite database is used to store file metadata and content. To narrow down cross-platform compatibility issues, this implementation will be developed only for Linux operating systems.

[3] The PreTFS source code is available at https://github.com/arsmoriendy/PreTFS

SQLite is chosen as the database because of its small footprint (~900-1500 kB), lack of dependencies, reliability, and fast performance [18]. Besides, as a relational database, SQLite complements well with how the proposed file system would be structured. For example, a prefixed directory might have multiple relationships (one to many) with many tags. This would be harder to implement within non-relational databases, such as NoSQL databases. Figure 3 describes the tables used within this implementation. To interact with the SQLite database in Rust, we will use a general-purpose SQL API called sqlx. Write-ahead logging (WAL) mode will be enabled for every connection in order to support read and write concurrency to the database.

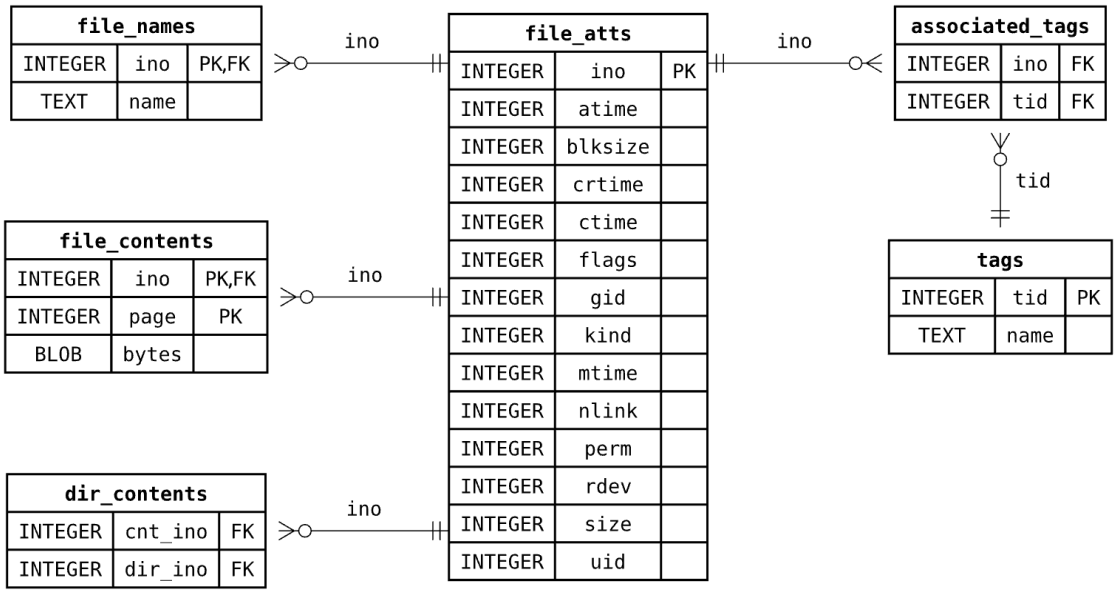

Figure 3. Entity Relation Diagram (ERD) for the SQLite tables

On the other hand, a FUSE library is needed to interact with the FUSE kernel module. Whilst libfuse is the standard FUSE API in C, fuser will be used as a replacement for libfuse's low-level API to communicate with the FUSE kernel module using Rust. Creating a file system in fuser requires an implementation of the fuser::Filesystem trait, which roughly maps to the fuse_lowlevel_ops struct in libfuse. From 45 fuse_lowlevel_ops methods, 39 of them exist also in fuser::Filesystem. However, not all of them need to be implemented; only 13 of them are needed for the proposed file system implementation. Within the scope of the proposed file system, only 7 of those are relevant, while the rest are just standard file system operations, like reading from or writing to a file. Concisely,

we can classify similar relevant methods into 5 groups, corresponding to each operation in Table 3. Table 4 lists the mapping between user operations and file system methods.

Table 4. User operations and file system methods mapping

User operation	Method(s)	Implementation detail
Creating tags	mkdir	The mkdir method allows the user to create directories. This is relevant because creating tags is done by creating prefixed directories. However, creating nested redundant prefixed directories is not allowed. For example, /notes/#biology/#biology is not allowed, since the nested #biology prefixed directory is redundant.
Create file tag associations	mknod, rename	The mknod method is used to create new files, whilst the rename method is used to move or rename files, by changing or "renaming" the associated file path. The mknod method is used to create file tag associations by creating a new file inside a prefixed directory, whilst the rename method is used to create file tag associations by moving a file to a prefixed directory. The rename method is not exclusive to files; directories can also be renamed or moved using the same method. Note that renaming a prefixed directory to an unprefixed one will throw an error.
Update file tag associations	rename	The rename method can also be used to update file tag associations by moving or renaming the path of a file from having one prefixed parent to another. It can also be used to delete file tag associations by moving a file from a prefixed directory to an unprefixed one.
List tagged files	readdir, lookup	The readdir method is probably the most important method within this implementation. It is used to list the inodes and names of children in a directory. On unprefixed directories, the readdir method works

User operation	Method(s)	Implementation detail
		relatively simply by querying the dir_contents database table. However, the readdir method doesn't work as trivially on prefixed directories. It needs to list nested prefixed directories by also querying the dir_contents database table. On top of that, it must list all unprefixed directories and files associated with the prefixed parent directory. Figure 4 is an example of a query that does just that. Since the readdir method only exposes the inodes and names of a directory's children, the lookup method is subsequently used to get correlated attributes for each child. The lookup method also uses Figure 4 on a prefixed directory to fetch associated children attributes based on their names.
Delete tags	rmdir, unlink	The rmdir method is used to remove directories, whilst unlink is used to remove files. The unlink method is called as such because it decrements the nlink attribute of a file (i.e., the number of hard links to a file). Once a file has zero hard links, it will be removed. To completely remove a tag, the user must remove all associated files and directories. One caveat is that calling rmdir on a prefixed directory will not remove its children4.

To facilitate 5 of the 7 relevant methods described in Table 4 (i.e., lookup, readdir, rmdir, unlink, and rename), a common and crucial database query pattern arises. These queries are used to fetch attributes of a file associated with two or more tags. Figure 4 is an example of such a query. The prominence of these queries is present in methods that

[4] The corresponding command for calling the rmdir method in Linux is the appropriately called rmdir; this is not to be confused with rm -r, which removes a directory and its contents recursively.

require reading the contents of a prefixed directory. Because of the length and many uses of these queries, a helper function is created within the implementation to reduce code duplication. This helper function was called chain_tagged_inos, and an algorithmic pseudocode notation of this function is shown in Algorithm 1. For example, Figure 4 can be easily created by calling Algorithm 1 with input q as "SELECT * FROM file_attrs WHERE ino IN", and T as $\{1,2\}$.

Algorithm 1: Pseudocode for the `chain_tagged_inos` helper function

Input: initial query q, tag set T with length n

FOR $i = 0$ **TO** n
  Push "`SELECT ino FROM associated_tags WHERE tid =` T_i" into q
  IF i **IS NOT** n
    Push "`AND ino IN (`" into q
FOR $i = 1$ TO n
  Push ")" into q

```SQL
SELECT * FROM file_attrs WHERE ino IN (
  SELECT ino FROM associated_tags WHERE tid = 1 AND ino IN (
    SELECT ino FORM associated_tags WHERE tid = 2
  )
)
```

Figure 4. An example SQLite query for listing the attributes of files associated with tags with tid 1 and 2

To conclude the implementation section, Figure 5 provides a simple running example for a write operation that spans the user request through PreTFS, SQLite, and the rest of the kernel space components. Below are explanations for each step of the process.

1) The user triggers a write() system call for a file located within the PreTFS file system. This request is sent to the VFS.
2) The VFS hands the request off to the corresponding file system where the file lives. In this case, since the responsible file system is PreTFS — which is a FUSE file system — the FUSE kernel module will handle it.
3) The FUSE kernel module forwards the request to PreTFS via system calls.
4) PreTFS updates the relevant tables within the SQLite database.

5) Since SQLite databases take the form of a file, a request to update the database file is sent to the VFS via system calls.
6) The VFS routes the request towards the file system holding the database file (i.e., btrfs).
7) Finally, btrfs writes the updated database file to the disk partition.

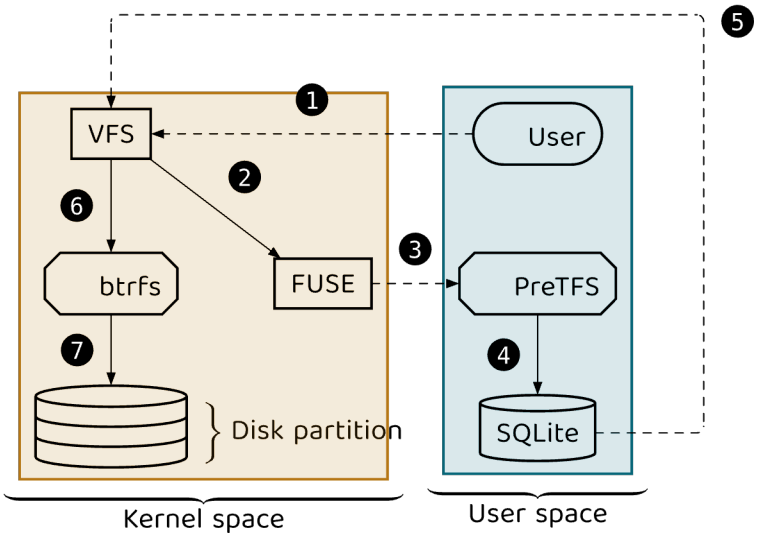

Figure 5. Diagram of how a write operation is handled. Dashed lines represent system calls.

3. RESULTS AND DISCUSSION

3.1. Use Case Demonstration

In this section, a synthetic scenario of Figure 2 will be implemented within PreTFS. Figure 6 shows the hierarchical structure of this experiment. The bracketed numbers (e.g., [1]) show the corresponding inode of each file. Conceptually, the demonstration confirms that PreTFS can represent the proposed design correctly. It shows how a single inode (biochemistry.md) can surface in every prefixed path that matches its semantic tags, such as /notes/#biology and /notes/#biology/#chemistry, without duplicating the underlying entry. The consistent inode numbering across these directories illustrates that the tag intersections are derived simply by nesting prefixed directories, not by copying or renaming files. Simultaneously, the example preserves the expectation that unprefixed directories like /notes/assignments behave identically to a traditional hierarchy, ensuring the overall structure remains interoperable with conventional tools.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

```
[      1]  notes/
├── [      4]  assignments
│   └── [      5]  dissertation.tex
├── [      2]  #biology
│   ├── [      8]  assignments
│   │   └── [      9]  taxonomy.md
│   ├── [      7]  biochemistry.md
│   └── [      6]  #chemistry
│       └── [      7]  biochemistry.md
└── [      3]  #chemistry
    ├── [     10]  #attachments
    └── [      7]  biochemistry.md

7 directories, 5 files
```

Figure 6. Result of running the command tree --inodes notes/

3.2. Performance Analysis

Whilst PreTFS implementation in this paper does not primarily focus on performance, this section will discuss the benchmarks done on PreTFS. Figure 7 shows metadata-related benchmarks, whilst Figure 8 shows IO-related benchmarks. In addition, Table 5 displays a comparison of median values for the benchmark. Within the benchmarks, the SQLite database file for PreTFS resides within the same btrfs file system that it is compared to. For each benchmark iteration (i.e., the x-axis), 100 runs are measured. Every set of diagrams in a row is measured using the same command; depending on the iteration, as shown in Table 6. Caching is disabled to ensure consistency between benchmark runs. This is done in two parts: (1) kernel caches are dropped prior to every run by running "sync; echo 3 > /proc/sys/vm/drop_caches", and (2) SQLite page and statement caching is disabled by setting the cache_size "PRAGMA" and the statement_cache_capacity connection option to 0.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

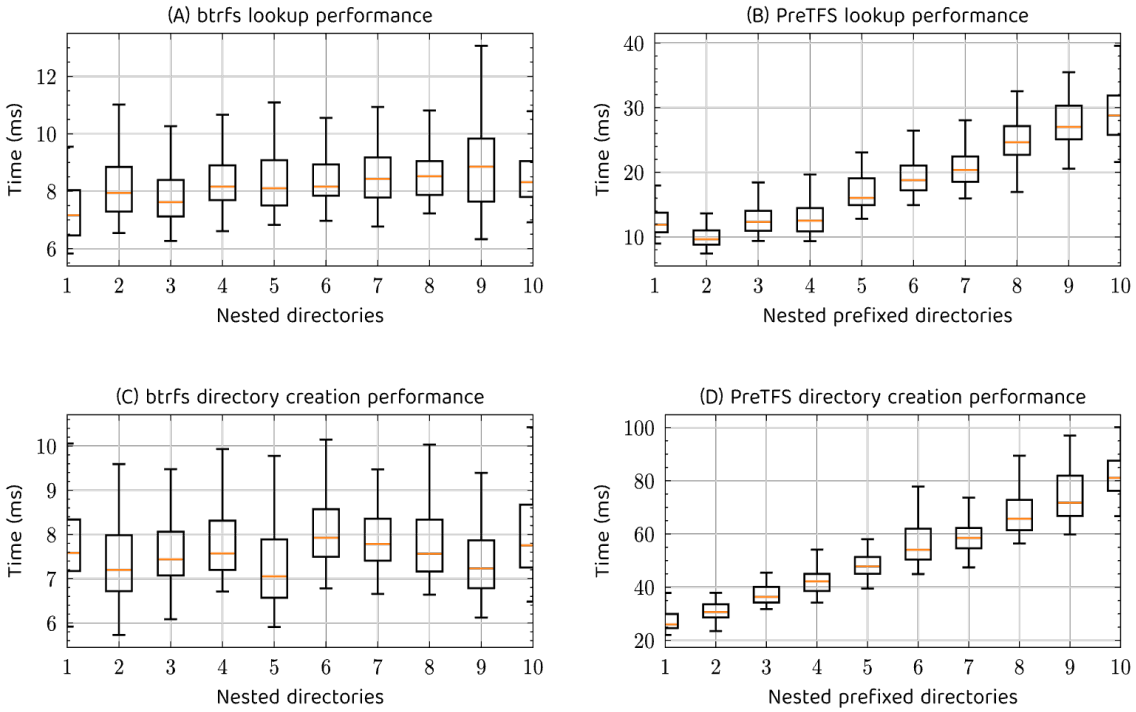

Figure 7. File metadata related benchmarks of btrfs and PreTFS

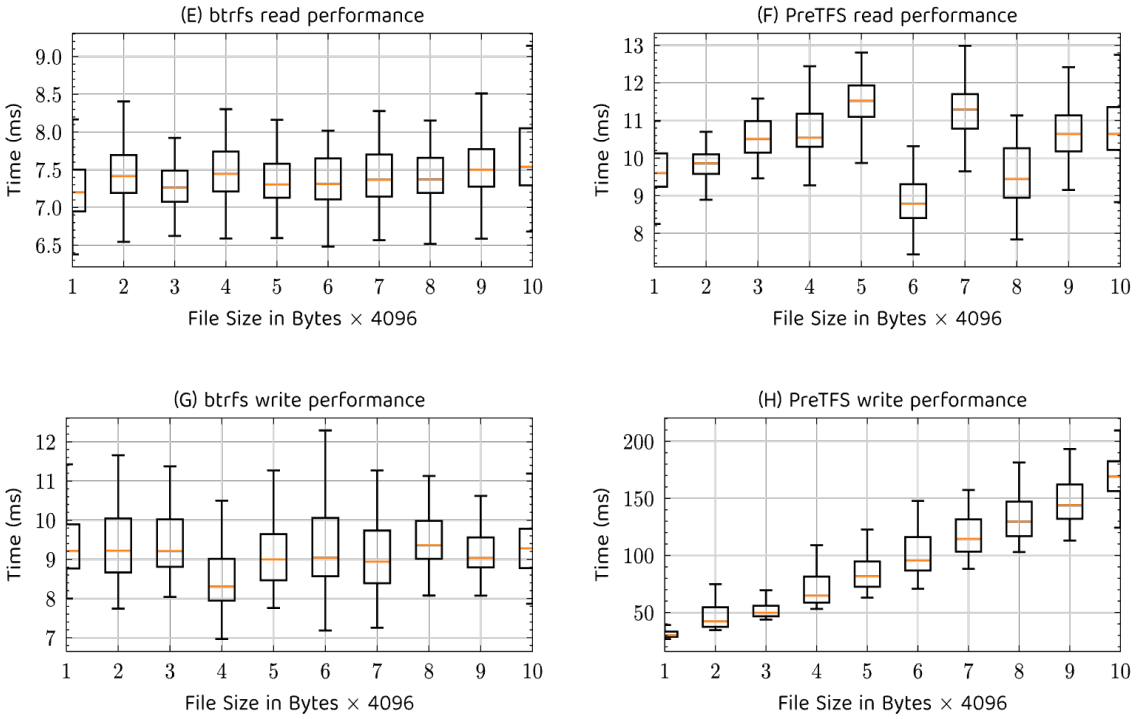

Figure 8. File content related benchmarks of btrfs and PreTFS

Table 5. Comparison of benchmark results between PreTFS, btrfs, and their overhead.

Benchmark	PreTFS			btrfs			Overhead		
	min	max	avg	min	max	avg	min	max	avg
Lookup	10ms	29ms	18ms	7ms	9ms	8ms	2ms	20ms	10ms
Directory creation	26ms	81ms	51ms	7ms	8ms	8ms	18ms	73ms	44ms
Read	9ms	12ms	10ms	7ms	8ms	7ms	1ms	4ms	3ms
Write	30ms	169ms	92ms	8ms	9ms	9ms	21ms	160ms	83ms

Table 6. Commands ran for each benchmark

Benchmark	Command
Lookup	ls mountpoint/{nested-directories}/file
Directory creation	mkdir -p mountpoint/{nested-directories}
Read	cat mountpoint/file
Write	cat /dev/random | head --bytes {file-size} > mountpoint/file

3.3. Performance Summary

The benchmarks indicate that PreTFS introduces measurable overhead across lookup, creation, read, and write operations, mainly due to FUSE syscall latency [19], [20] and additional SQLite queries for tag intersections and paged file contents. However, these costs remain bounded and predictable, and they trade off against stronger semantic flexibility and interoperability goals. Figure 7, Figure 8, and Table 5 reveal the following performance characteristics for each benchmark category:

1) Lookup performance**:** Diagrams (A) and (B) in Figure 7, along with the first row of Table 5 show PreTFS has linear lookup times (~10-29ms) depending on nesting level, compared to btrfs (~8ms). This results in an average overhead time of ~10ms.
2) Creation performance**:** Diagrams (C) and (D) in Figure 7, along with the second row of Table 5 demonstrate linear scaling for PreTFS (~26-81ms) versus constant time for btrfs (~8ms). Resulting in an average overhead of ~44ms.
3) Read performance**:** Diagrams (E) and (F) in Figure 8, along with the third row of Table 5 display non-linear yet variable results for PreTFS (~10-13ms). On the

other hand, btrfs maintains a more consistent performance (~7ms). Resulting in the smallest average overhead of ~3ms.

4) Write performance**:** Finally, diagrams (G) and (H) in Figure 8, along with the fourth row in Table 5 highlight the biggest performance contrast between the two file systems. The benchmarks show that PreTFS linearly scales for each size increase (~30-169ms), whilst btrfs is indifferent between write sizes (~9ms). The overhead for this benchmark is also the highest at ~83ms.

Acceptable use cases include research or documentation collections that need multiple semantic views, environments where metadata richness matters more than raw throughput, and workflows that already tolerate FUSE latency because they mainly serve human users rather than bulk automated workloads. Unacceptable use cases are latency-sensitive bulk transfers, database servers that demand kernel-level throughput, storage for larger file sizes like modern video games, or where the added latency overhead would conflict with strict service-level objectives.

3.4. Discussion

The proposed file system design in this paper delivers a cohesive semantic experience, reliably surfacing prefixed files through standard utilities and giving users immediate organizational clarity without leaving the familiar hierarchy. Furthermore, the PreTFS implementation proves the feasibility of such a design. And finally, benchmark evaluations show that while there may be room for performance improvements, the proof-of-concept implementation suffices in use cases where metadata structure is pertinent.

One example implication of this paper is that the proposed file system may help in the digitalisation of personal information management or PIM systems [21]. A recent survey [7] shows gaps between actual and ideal PIM-related behaviors. Whilst the study shows that most of the gaps can be attributed to the user's ineffective usage of already existing PIM features, tagging stands out as the most prevalent underused feature. This may be caused by the lack of native tagging support in existing file systems, which is what this paper tries to solve. Still on the subject of PIMs, the proposed file system in this paper may aid file categorization described in the personal information organization process

(PIOP) [22] by allowing scenarios where the user evaluates that a file should reside in multiple directories. The PIOP paper [22] also states that one of the ways users categorize their files is by identifying the file type through the file extension. However, another research indicates that many users lack a functional understanding of file extensions [23], suggesting that semantic tagging provides a more accessible retrieval mechanism. To wrap up the discussion on PIMs, a recent review on the future of PIMs [5] correctly states that file system design has not evolved in adoption for decades, and that support for interlinking is lacking. The file system proposed within this paper provides interlinking in the form of semantic categorization, and may be a candidate for the future of PIMs.

While the design proposed in Section 2.1 establishes a robust framework for hierarchical interoperability, certain architectural choices necessitate trade-offs. Although the system successfully manages the majority of standard operational scenarios, specific edge cases regarding data persistence and naming conventions warrant further discussion:

1) Superset duplicate handling

A significant challenge arises when distinct files share the same name but possess overlapping tags. For instance, if note.md exists in /notes/#biology/ and a different note.md exists in /notes/#biology/#chemistry/, both theoretically belong in the /notes/#biology view. However, since standard file systems prohibit duplicate filenames within a single directory, an ambiguity arises. The current implementation resolves this by displaying only the first file instance retrieved from the database. While automated renaming strategies (e.g., prepending numerical identifiers like 1-note.md) were considered, they were rejected to avoid conflicts with existing user-defined naming conventions.

2) Persistence of orphaned views

A dissonance exists between standard directory removal and the system's semantic logic. When a user deletes a tag-directory (e.g., /notes/#biology), its subdirectories (e.g., assignments) are removed from the immediate hierarchy. However, their records persist within the database, effectively becoming "orphaned" from the view. While implementing a "pruning" mechanism to scrub these orphans is possible, the design prioritizes data safety. Consequently, deletion operations on tag-prefixed directories are strictly scoped

to the view model, ensuring that modifying the navigational structure does not inadvertently destroy the underlying file content.

3) Optimal prefix

The choice of the prefix delimiter presents a trade-off between visual distinctiveness and shell compatibility. While the default hash symbol effectively distinguishes semantic tags from standard directory names, it conflicts with shell comment syntax. Alternative delimiters (such as! or &) pose similar challenges regarding history expansion or background process execution. To address this, PreTFS supports configurable multi-character prefixes (i.e., string prefixes). This allows the namespace delimiter to be extended beyond a single character, significantly reducing the probability of collision with both shell interpreters and existing file naming conventions.

4) Intersection exclusivity

The hierarchical mapping of tags inherently restricts the system to a single set operation. In the proposed design, prefixed directory nesting denotes the intersection of tags (i.e., the ∩ operator, filtering for files containing all specified tags). While the architecture could effectively model tag unions (i.e., the ∪ operator, aggregating files containing any of the tags), the linear nature of directory traversal prevents the simultaneous application of both intersection and union operators within the same path structure.

Finally, for future work, performance could be optimized through several strategies. One such way is by handing off file storage to the native file system, which might grant native read/write performance. On the other hand, metadata operations within a file system aren't usually performance-intensive. However, metadata operations account for ~80% of all file system operations [24], and improving their performance might yield significant results. There are many ways to improve metadata operations, such as optimizing *dentry* caches [25], among others. Further research may also focus on migrating the implementation entirely to a native kernel file system. There also exist recent FUSE-like innovations such as Direct-FUSE [26], which bypasses system calls between the kernel and user space. Another FUSE improvement called DeFUSE [27], might also improve performance by isolating file content related request within the kernel space, whilst still allowing metadata requests through the user space.

4. CONCLUSION

This paper presents the design, implementation, and evaluation of a tag-based semantic file system that uses a novel approach of using prefixed directories to represent tags. Its prefix-driven structure keeps conventional hierarchies intact while letting users express semantic intersections directly within the existing namespace. The PreTFS implementation proves that the proposed design can be realized. Performance evaluation reveals expected implementation overhead, with slower performance compared to a kernel space file system (i.e., btrfs), that may scale linearly in certain scenarios. These trade-offs are inherent to the FUSE architecture, in addition to an SQLite-backed approach, representing an acceptable cost for gaining better ease of development and installation compared to kernel space file systems. Comparison with existing tag-based file systems demonstrates that the proposed design provides a simpler, more straightforward approach to semantic categorization, whilst allowing conventional programs to interact with it seamlessly. Future work may include: (1) offloading file content to the underlying file system, (2) exploring kernel space implementation, and (3) investigating modern file system frameworks and architectures for improved performance.

REFERENCES

[1] H. tom Wörden, F. Spreckelsen, S. Luther, U. Parlitz, and A. Schlemmer, "Mapping hierarchical file structures to semantic data models for efficient data integration into research data management systems," *Data*, vol. 9, no. 2, Art. no. 24, 2024, doi: 10.3390/data9020024.

[2] Y. Wang, W.-Q. Jia, D.-J. Jiang, and J. Xiong, "A survey of non-volatile main memory file systems," *J. Comput. Sci. Technol.*, vol. 38, no. 2, pp. 348–372, 2023, doi: 10.1007/s11390-023-1054-3.

[3] Y. Yang, Q. Cao, J. Yao, Y. Dong, and W. Kong, "SPMFS: A scalable persistent memory file system on Optane persistent memory," in *Proc. 50th Int. Conf. Parallel Process. (ICPP '21)*, Lemont, IL, USA, Aug. 2021, doi: 10.1145/3472456.3472503.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

[4] H. Leblanc, N. Taylor, J. Bornholt, and V. Chidambaram, "SquirrelFS: Using the Rust compiler to check file-system crash consistency," *ACM Trans. Storage*, vol. 21, no. 4, Nov. 2025, doi: 10.1145/3769109.

[5] A. Dix, "The future of PIM: Pragmatics and potential," *Hum.-Comput. Interact.*, vol. 41, no. 2, pp. 126–153, 2026, doi: 10.1080/07370024.2024.2356155.

[6] T. Habermann, "Metadata life cycles, use cases and hierarchies," *Geosciences*, vol. 8, no. 5, Art. no. 179, 2018, doi: 10.3390/geosciences8050179.

[7] L. Alon and R. Nachmias, "Gaps between actual and ideal personal information management behavior," *Comput. Hum. Behav.*, vol. 107, Art. no. 106292, 2020, doi: 10.1016/j.chb.2020.106292.

[8] S. R. Mashwani and S. Khusro, "360° semantic file system: Augmented directory navigation for nonhierarchical retrieval of files," *IEEE Access*, vol. 7, pp. 9406–9418, 2019, doi: 10.1109/ACCESS.2018.2890165.

[9] O. Bergman, T. Israeli, and Y. Benn, "Why do some people search for their files much more than others? A preliminary study," *Aslib J. Inf. Manag.*, vol. 73, no. 3, pp. 406–418, 2021, doi: 10.1108/AJIM-08-2020-0250.

[10] O. Bergman, T. Israeli, and S. Whittaker, "Factors hindering shared files retrieval," *Aslib J. Inf. Manag.*, vol. 72, no. 1, pp. 130–147, 2020, doi: 10.1108/AJIM-05-2019-0120.

[11] P. Englefield and R. Beale, "How helpful is it to organize personal information by activity?," *Behav. Inf. Technol.*, early access, 2025, doi: 10.1080/0144929X.2025.2560551.

[12] S. R. Mashwani and S. Khusro, "The design and development of a semantic file system ontology," *Eng. Technol. Appl. Sci. Res.*, vol. 8, no. 2, pp. 2827–2833, Apr. 2018, doi: 10.48084/etasr.1898.

[13] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and J. W. O'Toole, "Semantic file systems," in *Proc. 13th ACM Symp. Oper. Syst. Principles (SOSP '91)*, Pacific Grove, CA, USA, Oct. 1991, pp. 16–25, doi: 10.1145/121132.121138.

[14] S. Bloehdorn, O. Görlitz, S. Schenk, M. Völkel, and F. Karlsruhe, "TagFS—Tag semantics for hierarchical file systems," in *Proc. 6th Int. Conf. Knowl. Manag. (I-KNOW '06)*, Graz, Austria, 2006, pp. 6–8.

[15] N. Albadri and S. Dekeyser, "A novel file system supporting rich file classification," *Comput. Electr. Eng.*, vol. 103, Art. no. 108081, 2022, doi: 10.1016/j.compeleceng.2022.108081.

[16] N. Albadri, "Attributes or tags for files? AttFS: Bringing attributes to the hierarchical file system," *Mars J. Tek. Mesin Ind. Elektro Ilmu Komput.*, vol. 3, no. 1, pp. 120–138, Jan. 2025, doi: 10.61132/mars.v3i1.607.

[17] T. Mkrtchyan *et al.*, "dCache: The storage system of choice for data-intensive applications," *Comput. Softw. Big Sci.*, vol. 9, no. 1, Art. no. 20, 2025, doi: 10.1007/s41781-025-00152-5.

[18] K. P. Gaffney, M. Prammer, L. Brasfield, D. R. Hipp, D. Kennedy, and J. M. Patel, "SQLite: Past, present, and future," *Proc. VLDB Endow.*, vol. 15, no. 12, pp. 3535–3547, Aug. 2022, doi: 10.14778/3554821.3554842.

[19] B. K. R. Vangoor *et al.*, "Performance and resource utilization of FUSE user-space file systems," *ACM Trans. Storage*, vol. 15, no. 2, May 2019, doi: 10.1145/3310148.

[20] S. Miller *et al.*, "High velocity kernel file systems with Bento," in *Proc. 19th USENIX Conf. File Storage Technol. (FAST '21)*, Feb. 2021, pp. 65–79.

[21] Y. Miyata *et al.*, "Personal information management practices among the general public: Analysis of questionnaire survey results in U.S. and Japan," *Aslib J. Inf. Manag.*, early access, 2025, doi: 10.1108/AJIM-10-2024-0839.

[22] K. E. Oh, "Personal information organization in everyday life: Modeling the process," *J. Doc.*, vol. 75, no. 3, pp. 667–691, 2019, doi: 10.1108/JD-05-2018-0080.

[23] P. Stephens and M. McGowan, "File management: Student knowledge of file type extensions," *Issues Inf. Syst.*, vol. 21, no. 3, pp. 236–244, 2020, doi: 10.48009/3_iis_2020_236-244.

[24] H. Dai, Y. Wang, K. B. Kent, L. Zeng, and C. Xu, "The state of the art of metadata management in large-scale distributed file systems—Scalability, performance and availability," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 3850–3869, 2022, doi: 10.1109/TPDS.2022.3170574.

[25] N. Y. Song, H. Kim, H. Han, and H. Y. Yeom, "Optimizing metadata management in large-scale file systems," *Cluster Comput.*, vol. 21, no. 4, pp. 1865–1879, Dec. 2018, doi: 10.1007/s10586-018-2814-7.

[26] Y. Zhu *et al.*, "Direct-FUSE: Removing the middleman for high-performance FUSE file system support," in *Proc. 8th Int. Workshop Runtime Oper. Syst. Supercomput. (ROSS '18)*, Tempe, AZ, USA, 2018, doi: 10.1145/3217189.3217195.

ISI Journal of Information Systems and Informatics
Published By Asosiasi Doktor Sistem Informasi Indonesia

[27] W. Yan, J. Yao, and Q. Cao, "Defuse: Decoupling metadata and data processing in FUSE framework for performance improvement," *IEEE Access*, vol. 7, pp. 138473–138484, 2019, doi: 10.1109/ACCESS.2019.2942954.