Enriching personal information management with document interaction histories

by
Karl Gyllstrom

A dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2009

Submitted for approval by:

David Stotts, Advisor

Craig Soules, Reader

Ketan Mayer-Patel, Reader

Diane Kelly, Committee Member

Mary C. Whitton, Committee Member

Abstract

KARL GYLLSTROM: Enriching personal information management with document interaction histories.
(Under the direction of David Stotts.)

Personal information management is increasingly challenging, as more and more of our personal and professional activity migrates to personal computers. Manual organization and search remain the only two options available to users, and both have significant limitations; the former requires too much effort on the part of the user, while the latter is dependent on users' ability to recall discriminating information. I pursue an alternative approach, where users' computer interactions with their workspaces are recorded, algorithms draw inferences from this interaction, and these inferences are applied to improve information management and retrieval for users. This approach requires no effort from users and enables retrieval to be more personalized, natural, and intuitive.

The *Passages* system enhances information management by maintaining a detailed chronicle of all the text the user ever reads or edits, and making this chronicle available for rich temporal queries about the user's information workspace. Passages enables queries like, "which papers and web pages did I read when writing the 'related work' section of this paper?", and "which of the emails in this folder have I skimmed, but not yet read in detail?" As time and interaction history are important attributes in users' recall of their personal information, effectively supporting them creates useful possibilities for information retrieval. I present methods to collect information about the large volume of text with which the user interacts, and use this information to improve retrieval. I show through user evaluation the accuracy of Passages in building

interaction history, and illustrate its capacity to both improve existing retrieval systems and enable novel ways to characterize document activity across time.

Before the Passages system, I developed two other systems with similar goals. *Confluence* extends an existing system that identifies task-based links among users' data through their being used at proximal points in time. For example, if a user frequently interacts with a report and a graph at the same time, those documents likely share a common task even though they may have no semantic relationship. Once such links are identified, they are applied when users issue search queries, expanding traditional, text-based results with other documents that share task-based links to those results. This creates a form of task-based retrieval which is independent of document semantics, and enhances users' ability to retrieve information. The *SeeTrieve* system extends this concept to trace the visible text in the GUI with which the user interacts and associate this with files whose accesses occur at proximal points in time. In addition to improving retrieval for users, it creates a form of automated, task-oriented tagging of files.

Acknowledgments

I thank David Stotts for his constant support and reassurance, especially in the time in which my future as a Ph.D. student was most uncertain. David saw in me a spark that he believed could make me a researcher, even when the first wave of qualification exams suggested otherwise. Not only did this help me stay afloat in the department, but it gave me the confidence to know I could do it. He is one of the main reasons I survived in graduate school.

My time at HP working with Craig Soules and Alistair Veitch was an amazing opportunity from which I learned a lot and for which I am extremely grateful. Craig was a critical influence on my work, and I highly respect his opinion and insight in research. Combined with his generosity with his time and guidance, it is very comforting to see him succeed in the research community, which gains much from his involvement.

My first experience with Diane Kelly involved her agreeing to read some of my work and providing insightful comments, amid the extremely busy schedule of a tenure-track professor, and with me being a stranger from a separate department. That event revealed a lot about her character. Diane is an excellent researcher who is also very open and giving. She sets an excellent example from which I have learned immensely, and is an asset to the research and academic communities.

I thank Ketan Mayer-Patel for his insight and enthusiasm, which drew me to him as a committee member even though our research interests did not obviously overlap. I thank Mary C. Whitton for her impeccable manuscript reviewing over the years, which

has always challenged me to (at least try to) be a better writer, as well as her rigorous insights into study design.

I thank my parents for all of their love and support, including reading various drafts of mine, however boring they now seem. I thank my wife Nilam for her eternal support which has given me so much strength. And to Nilam, I apologize for the difficulties of my studies which often forced me to be far from her. My cousin Daniel Gyllstrom deserves thanks for reading and offering comments on my work, as well as being a great person to share gripes with.

I thank my friends, which give me the fortunate inconvenience of being too numerous to list here. In particular, a number of them were crucial to my success at graduate school: Ted Kim, Youn Ok Lee, Andrew Leaver-Fay, Joe Fitzgerald, Scott McLin, Noyle Jones.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Every day, more of our personal and professional activities enter the digital world, creating a permanently growing amount of personal information to manage. Nearly every photograph, work document, music album, and correspondence we use over the remainder of our lives will be stored in digital form. As storage technology is now practically infinite, and high speed networks and mobile devices make our data ubiquitous, challenges in personal information management have outgrown the technical domain. Now, we need imaginative and innovative approaches to helping users manage the overwhelming volume of personal data they use and generate. This is the problem of supporting effective *personal information management*.

Current tools for personal information management are not sufficient. Users are generally required to manually organize their own information. Filesystems, the primary medium for personal data management today, provide a single directory hierarchy, forcing users to constrain personal data sets to a limited classification system, even when that data contain elaborate and complex relationships. Though other approaches have attempted to support a more free-form classification system, they still require manual classification by users. As research shows, users find manual classification to be more than a simple annoyance; rather, new files often defy an immediate classification, and

taxonomies must be continuously updated as users' conceptualization of their personal information evolves (e.g., [39, 56]).

On the other hand, personal search tools such as *Google desktop* enable users to defer classification and instead use their own recollection of file contents to drive search queries when the file is again needed. And yet, search systems do not solve the organization problem precisely because they require that users remember at least some exact details of the file's text in order to formulate effective queries. Users often fail to recall details of files' contents even after short periods since last encountering them; consider how difficult this recollection would be for files which were last accessed months, years, or even decades previously.

Even in the case of properly recalled details, personal search systems cannot benefit from the same approaches that make web search systems like Google effective. The ability to infer the credibility of a page from how other pages link to it, a core ability of web search ranking, requires the community of web users to structure the web via the complex hyperlinks among sites. Without such structure, personal search relies on text, which alone is insufficient for high quality performance.

Users' documents are personal, so personal characteristics of information should be an emphasis of information management systems. In this thesis, I explore using certain qualities of the history of a user's interaction with his files as a pathway to the subsequent retrieval and management of those files. Interaction, in this context, includes attributes such as when, for how long, and within what task a file was used. Conceptually, these attributes are often closely aligned with the way in which users recall documents (e.g., [5, 25, 53]). Users tend to conceptually organize items according to overarching tasks and themes which are temporal in nature, and to apply these concepts in recollection. For example, a user may forget exact words within, or the filesystem location of, a paper they wrote, but may well remember the task it was

written for (e.g., the IUI conference), when the task occurred (e.g., last fall), and how the file was used (e.g., heavily during the task, rarely after). Though useful, current system support for such features is simplistic and superficial, if not lacking altogether. I address this through the design and implementation of three systems, each of which deals with a particular aspect of the problem. These systems are *Passages*, *SeeTrieve*, and *Confluence*.

The *Passages* system enables users to retrieve files by issuing temporal queries of novel richness and granularity. As time is an important and nuanced aspect of users' recollection of their personal information, supporting rich temporal queries is important for information management systems. Passages enables queries like, "which papers and web pages did I read when writing the 'related work' section of this paper?", and, "which of the emails in this folder have I skimmed, but not yet read in detail?" Passages works by recording text which appears in the active application window within the system graphical user interface, including when and for how long it remains visible. This timing information is then associated with the files which contain the text, as meta-data. Since the user interface is tightly coupled with users' actions than the file system, it provides rich and granular information about users' interaction with text, allowing Passages to record a detailed interaction history between users and files which would not be possible through other means (such as file-access timestamps, present in existing operating systems). In Chapter 5, I present how Passages collects and makes sense of the large volume of text with which the user interacts. I show through user evaluation the accuracy of Passages in building interaction history, and illustrate its capacity to both improve existing retrieval systems and enable novel ways to characterize document activity across time.

SeeTrieve is a personal document retrieval and classification system which abstracts applications by considering only the text they present to the user through the focused

window in the user interface. Timestamps document the points at which a given sequence of visible text goes in and out of view. Associating the window text which is viewed in the time surrounding a document's use, SeeTrieve is able to identify important information about the task within which a document is used. This context enables novel, useful ways for users to retrieve their personal documents. For example, a user may forget the location of a downloaded email attachment but remember text from the email itself; SeeTrieve would enable this user to issue a query on the email text and have it return the attachment as an additional result. Instead of just indexing file contents, SeeTrieve captures and indexes snippets of text displayed at the user-interface level by applications. Using temporal locality, it creates a mapping between these snippets and the files accessed while the snippets were displayed. This extends the traditional document search mechanism of a two-level mapping of terms to documents to a three-level mapping of terms to snippets to documents. Just as in a two-level index, SeeTrieve can use its three-level index to both classify documents – finding relevant terms from related text snippets – and retrieve documents, searching the index of text snippets and then following them to relevant documents. When compared to content-based systems, this task-based retrieval achieved substantial improvements in document recall. SeeTrieve is described in Chapter 4.

Confluence identifies task-based relationships among files by recording file access events in the operating system that result from users' file interactions through applications, and discovering which files are frequently accessed at proximal times. It extends an existing file search system, called Connections [53], by tracing user activity in the user interface in addition to the file system. As demonstrated by the Connections user study, proximal accesses are useful indicators that files share a task relationship [53]. The identified relationships are then applied in desktop searches, where users' search results are augmented with other files that share common tasks with those files. This

form of retrieval is useful in scenarios in which the user does not remember text from a file, but remembers other files that were used alongside the file. For example, they may remember the location of a paper they wrote but not recall the location of a file depicting a graph which was included in the paper. Confluence operates by tracing user-file interaction using events from two system layers: the file system and the graphical user interface. This two-layer approach was shown to be an effective approach to task detection through a user evaluation. I describe Confluence in Chapter 3.

Of these systems, Confluence was designed first, and Passages was designed last. Each of these systems were conceived, in part, by limitations in the previously implemented system. SeeTrieve was motivated by the inability of Confluence to capture activity occurring on files which have no local embodiment (e.g., a web-based email). Additionally, the recording of user's interaction with text in the UI is much more granular in terms of time. Where file accesses are discrete events, from which the duration of use cannot be inferred, the amount of time over which a window of text is visible in the UI has a precise start and end date, and consequently a duration of use.

A limitation of SeeTrieve is that its handling of UI text is simplistic. Viewed text is used as a pathway to later retrieval of files used at proximal times, but there is no attempt to connect the use of that text over time. For example, the viewing of a page of a PDF at different points in time is treated as two unrelated events. Passage addresses this limitation, allowing information about a user's interaction with a given sequence of text accumulate over time. The solution to this problem (as described in Chapterchapter:passages) has general applicability that enables Passages to solve other problems, such as determining the file whose contents are (partially) displayed in the user interface, and determining the provenance and flow of information between files in the file system.

1.1 Summary of results

Confluence was shown to improve the quality of task-based context building through two user evaluations. These evaluations produced results which showed a statistically significant improvement over an existing, similar retrieval system, which itself was shown to improve retrieval for users in a naturalistic study in which they searched for personal data [53].

SeeTrieve was shown to improve retrieval for users in a user evaluation in which users engaged in tasks in which they interacted with documents, and later attempted to retrieve those files. SeeTrieve employed a task-based retrieval that outperformed an existing, text-based search engine that reflects the state-of-the-art for users today.

Passages was shown to provide highly accurate tracing of users' interaction with documents. These results were compared to purely file-based tracing, which performed substantially worse by comparison. Combined with a performance evaluation in which the size and speed of Passages were shown to be practical for interactive systems, there is evidence that Passages would make a strong system for tracing users' document interactions.

1.2 Problem overview

Problems of capturing, representing, and applying information-interaction spans theoretical and practical realms, both of which are significant and require novel solutions. Theoretical problems include:

- forms of information to capture
- how to store and represent information
- how to apply this information in a way which improves personal information

management

Practical problems include:

- how to solve the theoretical problems within the limitations of entrenched technologies, like applications, filesystems and operating systems
- efficiency and scalability
- how to evaluate the effectiveness of solutions

The literature has demonstrated a need for improved system support of certain document attributes, many of which can be captured in a document's interaction history (e.g., [5, 25]). This thesis will focus on supporting two aspects of document history:

- capturing *when* and *for how long* a document is interacted with
- capturing the relationships among application window text and documents via a shared *task context*

1.2.1 Temporal characteristics of document interaction

Time is a natural axis along which we think about our documents. Almost any personal document is situated within a task (e.g., a conference paper) containing temporal qualities such as date (e.g., the deadline was in February), frequency (e.g., I still refer to it occasionally as I write my proposal), and duration of use (e.g., for most of January). Numerous studies report that the temporal history of a file's activity, such as when they last used it, are strong components of users' memories of their documents [5, 21, 22, 24, 35].

Currently, support for time exists in some form in many different applications. Filesystems reveal the creation and modification times of files to provide some context.

Email systems maintain records of when messages are sent and received. This intuitive integration of time has led many users to leverage email systems as tools for general document and task management [4, 12, 38]. Still, system support for time recording is simplistic and superficial, as detailed later.

1.2.2 Context

Context describes the fact that a piece of information (e.g., a file) exists as part of an abstract task which often incorporates other sources of information. We define task to mean activity toward satisfying a goal on the part of the user. This activity involves interaction with at least one document, and can long periods of time and be suspended and resumed. Tasks can have different scopes and even be contained within one another. For example, writing a paper for a conference is a task which can contain other tasks, such as writing the related work section. Both of these tasks can involve overlapping or disjoint sets of files. When viewed as a collective, the task informs the role of each individual piece of information within it, providing information about the file which is not contained within its contents. Research has shown task associations to be a common way for users to remember their files [5, 23]. For example, the process of writing a research paper often involves the synthesis of multiple files, such as text documents, graphs, and bibliography indexes. The meaning of a single file within this set, such as a file depicting a graph, may not be obvious without knowledge of its surrounding task.

Importantly, the task context of a file can be identified by observing the user's interaction with their information, and without requiring a semantic understanding of the task within which that interaction takes place nor the information contained within it. Specifically, patterns of temporally proximal use among files are a strong indicator of task commonality [43]. We will refer to *temporal locality* as the general

idea that events that occur at proximal points in time are likely to be related by a common task. While temporal locality is traditionally applied in caching systems, it nicely reflects intuition about how humans execute tasks, and has been applied with success in numerous settings involving user activity [43, 47, 53]. For example, if we see a person read three papers in a row, we can infer a relationship among those papers without knowing details of their contents (e.g., they may be part of a literature review, or assignments the reader is grading).

Context and time are important ideas in personal information management because they provide ways to make sense of unorganized information without requiring intervention from the user. The act of recalling information by its surrounding context is familiar for most people. For example, it may be difficult to remember the cryptic name of a previously downloaded email attachment, but easy to recall aspects of the email to which it was attached (e.g., *who* sent it to me? *when* was it sent?). In this example, the connection between items was explicit in that the attached file was associated to the email through the application. In numerous other scenarios, this association is missing.

1.3 Challenges in history tracing

Effectively capturing file access history requires observing and interpreting the user's interaction with their documents; this turns out to be difficult for a number of reasons. Historically, operating systems and applications have been the primary moderators of users' access to documents[1]. Consequently, it may seem as though these entities should be capable of capturing and retaining document interaction history. For example, by tracing file accesses by the user at the operating system level, we should be able to reconstruct the user's document interaction.

Unfortunately, such an approach is insufficient. The primary difficulty is that there

[1]The proliferation of web-based applications is challenging this.

is a large and growing rift between the user's personal information workspace (i.e., the documents with which they interact) and the systems into which this workspace manifests (i.e., the file system. Users' activities are not easy to trace for a number of reasons. To facilitate discussion on how this rift negatively affects the ability to build context, we will consider the interaction between user and document as a set of events occurring in one of two spaces: the *application-layer* and the *file layer*. The file layer comprises objects and activity within the file system, where a file exists as data identified by a unique path, and activity comprises a set low level file operations such as *read* and *write*. The application layer involves files and their activity as exposed through applications, such as typing text on to a window displaying a user's document (typically the user interface encompasses application-layer interaction).

For example, a user editing a resume for an internship application might use a text editor to add or change content. The application window displaying the resume is the application-layer document, and the user typing words into the document is a form of application-layer activity. Invisibly to the user, the text editor enables this interaction through file layer activity; specifically, reading the data of the file through the read system call and maintaining the state of the file within the filesystem through periodic write events. While this example presents a strong and natural coupling between spaces of the resume and the user's interaction with it, tasks and applications which are larger and more complex present cases where this coupling is much weaker, frustrating file-based context building. I focus on two categories of decoupling problems: *existential* and *causal*.

1.3.1 Existential

Existential problems refer to a decoupling between a document's abstract and concrete forms; specifically differences in the way a document manifests in the application and

file layers. With respect to the previous example, the user's document is — in the abstract — their most recent resume. Concretely, it may be a file on their laptop filesystem, or a file purely in memory (e.g., not yet saved), or a file on a remote server (e.g., Google Documents[2]). Existential problems occur when the user's documents are not instantiated in a manner that reflects their abstract representation; these present in three main ways: location, composition, and definition.

Location

Location decoupling occurs as the user's document workspace spreads beyond the user's personal computer to different systems, as is the case with remote filesystems. Due to the many advantages of distributed filesystems, application designers have embraced this paradigm and often insulate the user from knowing (or needing to know) where a file resides. For example, email clients differ on whether messages are saved locally or remotely. Note that location problems are not limited to distributed filesystems. Web-based email clients, for example, moderate email access but do not provide actual access to the files themselves (e.g., one could not choose to open the same email with a different client). Hence, where tracing or search systems may be able to work with traditional distributed file systems, they cannot work with web-based files because file access is not exposed through traditional means (or at all).

Composition

Composition decoupling occurs when a file's representation in the filesystem is not cleanly paired to its abstract representation. For example, in the past, email was saved as a directory of time-ordered files, each of which corresponded to a distinct message. Today's email client may adopt application-specific file management, such as merging

[2]Google Documents is an example from a recent generation of online productivity suites. In these systems, the application is used through the browser, and the files are stored on the remote server

all email into a single database file. The problem with the latter approach is that external management tools, such as desktop search engines, must implement awareness of each application-specific file approach. Further, document tracing is frustrated. Let us revisit the above example. As a user reads three different emails, we would expect to observe the application accessing three different files. If the client adopts a database approach, we will instead observe the same database file being accessed three times. While the database file technically encompasses the data accessed within this task, the granularity is far too coarse to be useful in activity tracing.

Definition

Definition problems occur as the criteria of what constitutes a file grow increasingly vague. Consequently, the traditional view of one's information space as a set of documents now fails to accurately model its use and organization in practice. Clearly, the document metaphor persists in some areas; for example, the reader of this proposal likely conceptualizes it as a single, coherent document. In other familiar contexts, this categorization is not so simple. For example, does an email constitute a document? What about an instant messenger conversation, a calendar appointment, or a history of calculator operations? A web page may seem like a meaningful unit of information, but website are often composites of different pages. For example, news aggregators display snippets from articles and provide links to the full article (Figure 1.1). In this case, perhaps the snippets themselves more meaningfully reflect files, with the web page itself playing the role of enclosing directory. The ability for applications to transcend the document metaphor has enabled novel forms of information interaction, but it complicates information management systems which have historically assumed a tangible file system that meaningfully reflects and encompasses the user's information space.

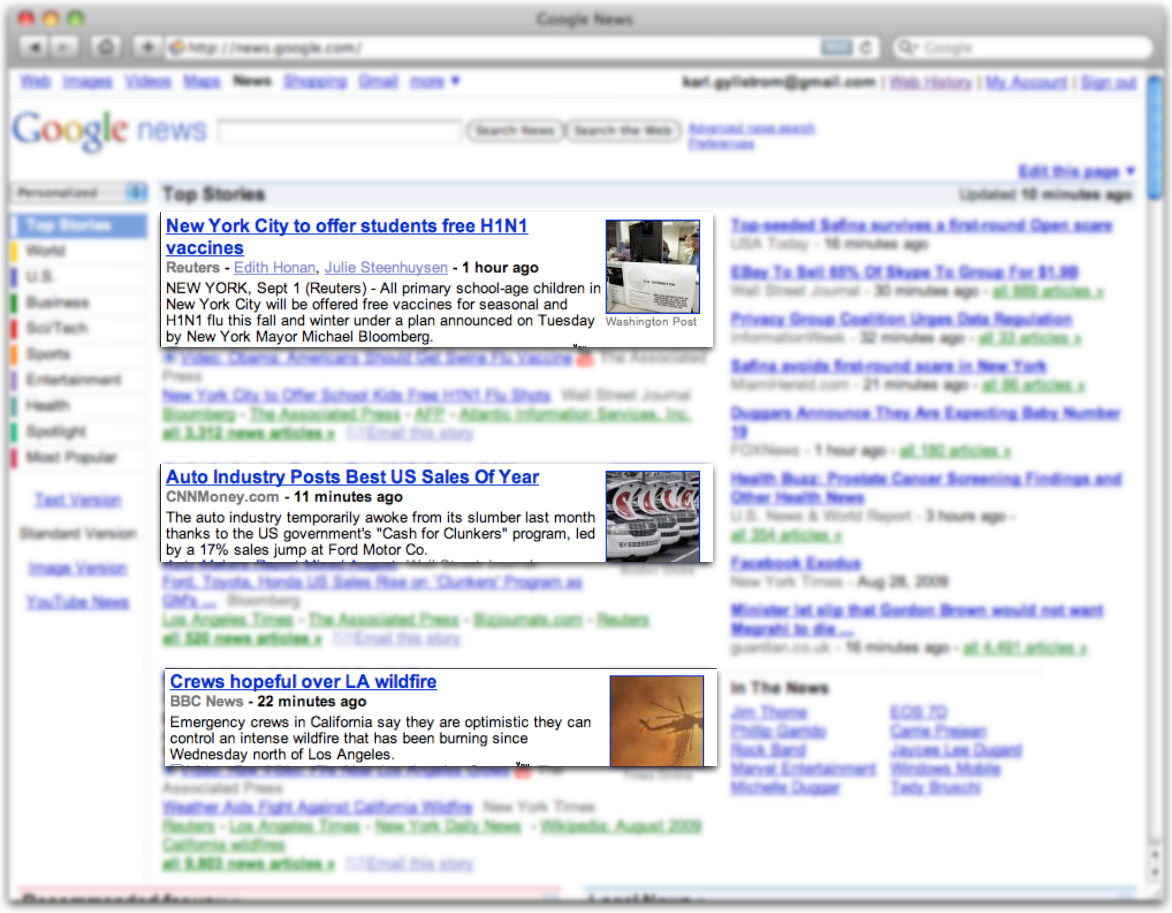


Figure 1.1: File definition problems for a news aggregator: though the web page shown would be treated as a single file by browsers, it actually serves more as a directory, with the snippets on the page more meaningfully representing files. Subsequent viewings of the page will likely contain different articles: each viewing reflects a viewing of new files rather than the same one.

1.3.2 Causal

Another problem with file layer abstraction is a separation between a user's conceptual document activity and the way in which that activity manifests within the filesystem. I call this class of problems *causal*, and it presents itself in two ways.

Origin

Origin refers to the disconnect between application activity and user activity. A user may have numerous application windows open while all but one are inactive or minimized. Because the user is only interacting with the visible window, it is likely that background activity performed by other active applications are acting without direction from the user. For example, a word processor may continue to generate automated save events on all of its open files even though the user has only recently interacted with one of the files.

Although users can only perform a single task at a time, the set of running applications may be performing any number of tasks in the background on the user's behalf. For example, while the user is editing a text document, other applications may be scanning for viruses, downloading new emails, playing music, etc. Any background file events interleave with the file events of the user's current task, making it difficult to distinguish between those which are and are not of user origin[3].

Finally, active user applications can also generate file events without application-layer activity. Many applications use a set of configuration and state maintenance files that are invisible or opaque to the user but necessary for the process' execution. Frequent accesses by the application to these files create the illusion that these files are part of the user's task when they are not. For example, a user opening a set of presentation slides considers only the slides themselves to be part of his context; however, the slide-authoring application may access configuration files, template files, libraries, etc., in the process of displaying the slide deck.

[3]For an example of the problem's scale, over the course of 5 minutes of the author editing this document, 1332 read/write events occurred involving 342 files on his machine. Only two documents were being actively used by the author; these documents experienced 7 read/write events between them.

Interaction

Interaction refers to the problem of knowing when, and for how long, a user interacted with data. Many recent systems, such as Watson [7] and others [3, 10, 19, 54], collect information about recently accessed files or web pages to recommend, or personalize, data on the web. Unfortunately, capturing "recently accessed" information can be complicated. For example, web browsers maintain caches of recently viewed sites, but they provide no information about how long a user viewed that site. Consequently, a link the user accidentally clicked through and returned from would be considered equally important to a page the user spent an hour reading. Inferring this information from the duration between their access times is not trustworthy; consider the case where one page is opened quickly after another, though in a different browser window. Should the user switch back to the previous window and spend a long time viewing the page, analysis of the browser history would indicate the wrong page was viewed for longer.

1.3.3 Summary

The general problem is that, though document interaction patterns do not vary significantly from the perspective of the user, it can vary widely from the system's perspective. The experience of working on simple tasks like editing a resume or checking email will be be largely unchanged from application to application and system to system, but the way in which that activity manifests in system-observable ways varies widely. Since document history tracing is limited by what the system observes, this is an important challenge.

The existential and causal problems comprise the difficulty of monitoring at the file layer. A potential alternative — at a potentially prohibitive loss of generality — is monitoring the applications themselves, using third party interfaces to determine the files with which the user interacts. However, requiring application support to identify

the user's file activity is not only problematic from an adoption standpoint; rather, it defies a fundamental need for applications to manage the presentation, interaction, and storage of information. In other words, applications would need to do much more than simply provide third party access to context-building programs (itself a massive task); rather, they would need to design (or re-design) their data management in accordance with a standardized document metaphor. Application specific approaches, then, should be ruled out when considering a comprehensive solution. Hence, we are left with two opposing needs: monitoring the user at a low enough level to obviate application support while accounting for the ways in which this level fails to fully reflect the user's information and activity. My thesis addresses this problem.

1.4 Thesis

Let us summarize the problem space:

- Modern retrieval and information management systems fail to support attributes of documents which are intimately related to the way in which people conceptualize and recall their documents. Temporal and task context are attributes of documents that have been specifically identified by the literature to be closely aligned with human recollection.

- Methods to passively monitor the user's interaction with their document space have been shown to enhance information retrieval, and literature has called for further innovation within this space.

- The rift between user and system activity challenges document interaction tracing.

Within this space, I propose the following thesis. **Modern, personal document management is enhanced by supporting rich document history, and creating solutions to the existential/causal problem space will improve support for document history collection.**

By monitoring user activity within the user interface — specifically, by following the user's application window focus events — we are able to mitigate causal problems by identifying activity within the file system which is more likely to be of user origin. This allows a more accurate identification of contextual relationships among files, and, in turn, will improve the performance of a document retrieval system for cases in which the application/file divide is significant. The Confluence system takes this approach and is described in Chapter 3.

By collecting the text which is made visible through the user interface and associating it with files which are accessed at proximal times, we can identify document interaction even in cases where a document's activity is obscured or nonexistent on the local file system (e.g., it is a web-based document). This provides a form of context which enables novel task-based retrieval, and will be shown to improve the performance of state of the art desktop search tools for certain classes of retrieval problems. The SeeTrieve system takes this approach and is described in Chapter 4.

By collecting the interaction history for UI text, we can support a form of document history tracing of more flexible document and temporal granularity. This will help address a need — as stated by recent research — for better support for representing temporal document activity, and can improve context identification. This contribution is accomplished by the Passages system, outlined in section 5.

Chapter 2

Background and related work

Personal information management (PIM) has evolved in both commercial and research domains. Unfortunately, due to entrenched operating systems whose file interaction support is difficult to bypass, ideas from the research domain are not easily ported to the commercial. This section attempts to reflect the divide in treatment of the subject.

2.1 What modern personal information management gets wrong

Modern document workspaces generally offer two approaches to information management: organization and search. Organization allows users to (1) place their files within a structure whose shape is (at least partially) under their control, and (2) assign attributes to files which reflect their meaning to the user. The most familiar example of the former is the hierarchical file system: a tree of folders and documents. The hierarchical file system affords some flexibility to the user in their ability to create arbitrarily nested directories containing files. While this structure allows users to organize their files according to semantic, temporal, or contextual relationships, it is limited by the rigid nature of trees, and awkwardly wedges the user's personal workspace into a larger

set of unfamiliar files, such as system libraries.

Some systems have addressed the limitations of the hierarchical file system by adding more flexible organizational tools. An example is tagging, the ability to annotate a document with unlimited user-generated terms, which can then be used to categorize, visualize, or search for documents. The improvement offered by approaches like this are limited to the physical act of organizing and do not address the core psychological problem: users have difficulty in organizing their documents simply because doing so is cognitively demanding [37]. Conceptual organization requires the user to devise a classification scheme within which not only current documents – but future, unknown documents – can be adequately situated. Documents may be created or accessed when the task of which they are a part is still amorphous, rendering classification premature. Future maturation of the task may require reclassification. Each attempt at organization involves a cost, even when many of the organized files may never be needed again. (Consider, for example, the web browsing experience if every visited page needed to be immediately placed in a relevant bookmarks folder.) For these reasons and more, organization is cognitively demanding for users and deferred whenever possible [33, 39, 40, 48, 56]. Hence, a large part of the information management challenge is inherent in the way users – as humans – conceptualize their information, and will likely persist in spite of increasingly sophisticated and flexible organization systems.

Search has gained popularity as an alternative to the organizational approach, offering users the ability to bypass manually categorizing documents [1]. Widely successful on the web, desktop search has experienced slow uptake due to relatively poor recall performance. This performance disparity can seem unintuitive as one's personal workspace is much smaller than the web. However, structural differences between the Web and one's personal filesystem play a critical role in this difference. Despite the

[1] See appendix A for a primer on search systems

chaotic quality often ascribed to it, the web has a highly organized structure from which powerful inferences can be drawn. A link between pages can indicate a semantic relationship (e.g., an article about socio-economic trends may link to census.gov), while a page with many incoming links from disparate sources is likely to be a credible source of information. PageRank, the core of the Google search algorithm, uses this structure to determine qualities like these [44]. Further, the volume of human-generated information allows other useful techniques to flourish. For example, a user-generated page in which an outgoing link is labeled with a misspelled anchor allows search systems to identify common misspellings and use these to improve the experience of future users who query with such misspellings.

This linking structure is built by the millions of web users who, simply by organizing their site through hyperlinks, provide rich information about how pages within their site are related to each other and the rest of the web. Successful Google queries, then, simply leverage the vast amount of organizing that the universe of web users have provided. For personal search to rival the performance of web search, we would need one of two impossibilities; users are neither able to (nor desire to) organize their own information as elaborately as the web, nor will others be able to (nor desire to) organize users' personal information for them.

Further, web based search is oriented to assist users in finding new information, rather than previously seen or lost information. This distinction is critical. Web pages are algorithmically deemed to be more credible resources if they are referenced by many other pages. Conceptually, they are popular and frequently accessed by others sharing a common information need. Adopting the same model to desktop search would mean users' searches for lost information would return highly ranked documents, which are actually those which are most popular and frequently used. This is the opposite of what is needed; users will more likely lose information which they rarely access.

The primary limitations with personal search are that it requires that users remember an exact term from the desired document, and that search results are ranked such that this document can easily be found in large result sets. As described previously, web search can leverage the community to help in both cases: using synonomy and context where exact terms are not recalled, and using structural information to infer authority or credibility for ranking. In the case where users recall nothing which a search engine needs to retrieve the lost item, they have little hope of retrieving their item.

Where web-based information retrieval strategies do not port to personal information management, there is also a failing of systems to properly address personalized aspects of information which could be advantageous in search contexts. Time of use, operational history[2] and surrounding task, for example, are strong memory triggers that search systems are unable to leverage in retrieval [5, 23, 24]. Even when systems do support a personalized attribute, they often do so in an overly technical way which obscures its meaning and usability [5]. Thus, a divide exists between system and user perceptions on these attributes. Consider the example of file size, which is a personal attribute of a file that is easily remembered the file's owner. Although this aspect is supported by filesystems, it is not presented intuitively: sizes are reported in terms of raw disk space (e.g., 5KB). Users think of text documents in conceptual terms, like "roughly 500 lines", "several paragraphs", or even relative to their other files of similar type or purpose (e.g., a dissertation is large compared to an article, but tiny compared to a video) [5].

A further and more pertinent example is with time. Though systems support some temporal qualities such as "most recent access", this attribute has a technical meaning which may not reflect the user's perception of actual use. A user may not consider

[2]events that occur upon the file, such as printing

quickly opening and closing a file to be reading it, and may confuse the term for involving a more in-depth interaction with a document. A virus-scanner may access the file and cause a technical file read that is unknown to the user. Further, meaningful aggregation of data is not possible.

2.2 Considering activity

Activity-aware systems mitigate the problem of cognitive overload by implicitly inferring organization and attributes from the user's document/information interaction. This derived structure can then be applied to document searches and browsing to facilitate information re-finding, improving the user experience by being more personalized and less demanding.

2.2.1 Activity-based memories

Activity is closely associated with human memory. Users tend to conceptually organize items according to overarching tasks and themes which are temporal in nature, and apply these concepts in recollection. For example, a user may forget exact words within or the filesystem location of a paper they wrote, but may well remember the task it was written for (a conference), when the task occurred (the months before the conference), and how the file was used (heavily during the task, rarely after).

Gonçalves et al. identified attributes about documents which users are most likely to remember [23, 24]. This took place over interviews in which users conceptually described personal documents with which they had previously worked. This information was used to inform the development of a narrative-based retrieval interface. Of many different abstract attributes that users recalled, both time of access and surrounding task were important.

Malone conducted a study of file organization in professional settings, finding a strong tendency of users to avoid organization and instead maintain loosely grouped piles of files [39]. Though limited to physical files, other studies have observed similar behavior in the digital realm [6, 56]. These piles often reflected frequency or recency of use, with this temporal data functioning not only to make data easier to find, but to serve a reminding role (e.g., a document needing hasty processing may be placed at the top of the pile). Though preferable to organizing due to the cognitive difficulty, piles break down once large enough, and would benefit from a more detailed temporal record of activity.

Blanc-Brude et al. refined this study with the intent to identify how effective users were at recalling these attributes (e.g., did they remember correctly the last time a document was accessed?) [5]. As with Gonçalves, they concluded that time and task were important attributes. They go on to specifically identify a need to enable file retrieval by related documents. They address a need for flexibility within time; allowing, for example, a temporal query to span a longer time (e.g., this file was read about a year ago, give or take a month). Finally, they advocate the potential retrieval benefits of a file's operational history, namely, events upon the file from the user such as printing or emailing.

Elsweiler et al. describe a diary study which examined the frequency and nature of memory lapses in the participants' daily lives [15]. They discuss the tendency for participants to use "retrieval journeys" in recalling forgotten information objects. For example, trying to recall where a paper is located, a user may first recall it by conference and use conference as the initial search criteria. They also offer several implications for the design of retrieval systems, including providing users the ability to "recreate the contexts in which objects had previously been accessed, used, or modified."

2.2.2 Research activity systems

Since Vannevar Bush's *memex* vision [8], much research has focused on the possibilities and technical challenges of recording and making sense of the user's interaction with information.

The spirit of activity-aware systems is perhaps best exemplified by the *Edit wear/Read wear* system [28]. The system was inspired by the observation that the migration of documents from the physical world to the digital world prevents the interaction with an artifact to manifest as wear upon it. This wear can offer clues about the artifact itself. For example, a physical cookbook exhibits forms of wear that reflect the way that book was used; it may more easily flip open to pages of recipes to which the reader frequently referred; and pages with special importance may contain dog-ears, stains, or liner notes. The authors speculate that recording and visualizing information about users' interaction with documents can enhance their recognition, intuitiveness, and usability. The Edit wear/Read wear system is a text editor which is enhanced by a sidebar which visualizes the amount of time a particular line has been in focus or manipulated (Figure 2.1).

Wexelblat applied the concepts of *Edit wear/Read wear* to a web browser [55]. In this work, users were able to visualize the paths formed by other users who traveled through the same websites. Though an interesting embodiment of activity-awareness, the work leverages the interactions of the community, rather than the individuals.

In a more macroscopic application of *Edit wear/Read wear*, Rekimoto [49] describes *TimeScape* a filesystem and interface that can be browsed along the time dimension, allowing one to see or restore a desktop composition from a previous date (see Figure 2.2). TimeScape augments the spatial desktop with the time dimension. Events such as file creation/modification, web pages surfed to, and emails received are recorded. The emphasis of this system was motivating the use of time as sharable metadata in

Figure 2.1: Edit wear/Read wear. Next to each line is a bar whose length reflects the amount of time in which the line has been in focus on the window, or the amount of editing which has taken place upon it.

applications from which a third-party system, like their visualizer interface, could operate, rather than offering novel methods for identifying document activity. Therefore, it is subject to the problems central to this thesis. Krishnan and Jones developed the *TimeSpace* system, a successor to TimeScape, which is oriented toward supporting user-defined tasks (as opposed to the entire file system) [34].

Fertig et al. present LifeStreams, a system that provides an alternative to the traditional desktop metaphor that replaces spatial organization by a set of temporal document streams, within which documents are placed automatically based on creation and use dates [17]. The theory is that work is naturally time-ordered, so a system which makes this ordering explicit and apparent makes document management more intuitive. This work has inspired many temporal systems, but itself employs a radical new desktop paradigm, rather than a practical extension to existing systems.

DejaView [36] records the user's desktop state (e.g., filesystem snapshots, active

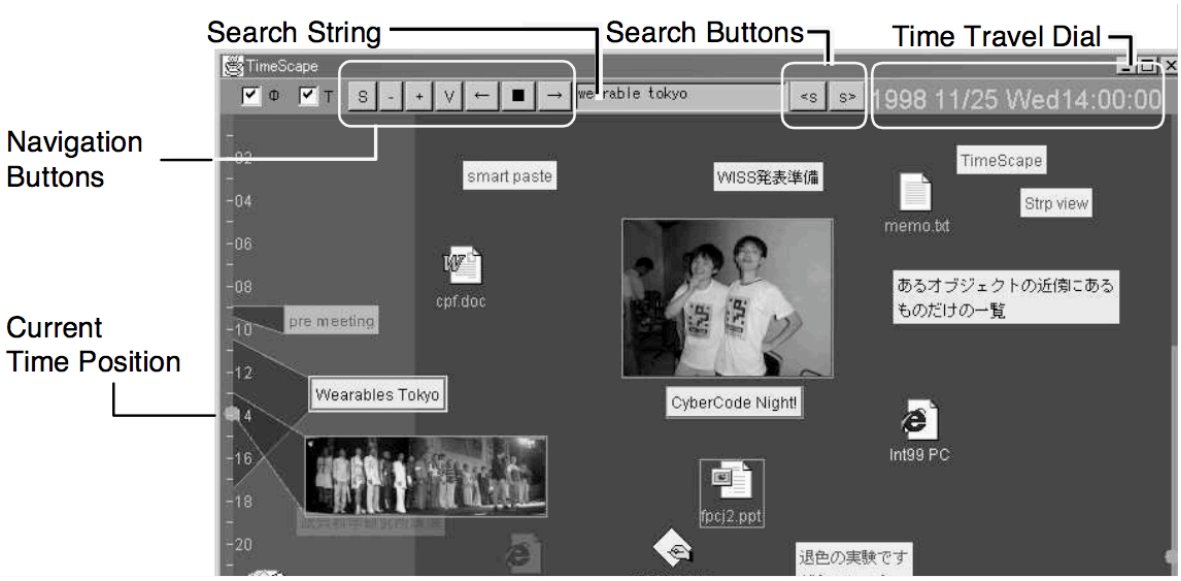


Figure 2.2: Timescape: a visualization of a desktop in which time can be changed. Previous dates show the contents and layout of files as they existed at that date. The timeline shows a visualization of file events.

application states, and screen contents) over time, allowing the user to search for and restore previous states. The system instruments the operating system to record file and network events, and instruments the graphical interface to record screen contents.

The *MyLifeBits* project aims to support lifetime personal information stores by retaining and managing as much of the user's personal information as possible, including all the documents with which they ever interact as well as video recordings and other media [20].

2.2.3 Task context

Task context describes the fact that a piece of information (e.g., a file) exists as part of an abstract task which often spans other sources of information. When viewed as a collective, the task informs the role of each individual piece of information within it, providing information about the file which is not contained within its contents. Research has shown this to be a common way for users to remember their files [5, 23].

For example, the process of writing a research paper often involves the synthesis of multiple files, such as text documents, graphs, and bibliography indexes. The meaning of a single file within this set, such as a file depicting a graph, may not be obvious without knowledge of its surrounding task.

In this section, I describe research which has used task context as a means to aid users in organizing their information. I split the work into three main branches. First, user-provided context systems support task context while not identifying it automatically. Content-based systems use the text of a user's history or activity to establish context. Finally, inferred context applies the user's document interaction in automatically building context.

User-provided context

UMEA[30] and TaskTracer [11] organize data according to discrete tasks that the user explicitly defines. UMEA presents a workspace manager through which users define their tasks, such as writing a proposal, and allows them to specify which task is currently active. In the background, the UMEA monitors Microsoft COM events to acquire traces of data objects accessed through Microsoft applications (e.g., URLs accessed by Internet Explorer or files created by Microsoft Word), and saves each data object as part of the active task. Later, the user may retrieve the set of collected data objects for any task through the UMEA interface. TaskTracer extends UMEA with a richer tracing infrastructure and provides the framework for new application types to have custom importers. These systems enable a form of automated organization, since once the user specifies the current task, all files accessed afterwords are automatically labeled as being part of that task.

Haystack [31] is an information management tool through which users can collect disparate data objects (e.g., emails, web pages, personal documents) into a single con-

textual grouping. Users can also specify the relationships among various data objects and groups, allowing a rich contextual framework. While per-application adapters can be used to enhance information about relationships, the primary focus is on providing flexibility to the user.

The biggest limitation of user-provided context is the requirement that users actively participate in task labeling, either via explicitly labeling the current task or by explicitly delineating relationships among data objects according to task. This contributes cognitive load which damages the ease of use and likelihood of uptake of these systems.

Content-based context

Content-based context uses semantic-based algorithms to derive relationships among documents and information. It is an appealing approach to task context because it does not require users to explicitly delineate relationships or to label and organize files according to their enclosing task.

Numerous approaches have been taken to add personalization to web searches. Consider the web search query "apple": it may return web pages pertaining to either the fruit or the computer company. Personalization, through user profile building, may reduce ambiguity. Some approaches create profiles based on information available in the user's browser cache [3, 54], URL history [19], or the contents of their computer [10]. These profiles are used to create a semantic model of users' tasks or interests, and the results of future web queries are tailored to more adequately reflect that model.

A number of systems use the contents of the user's activity to recommend related material. Watson uses application adapters to draw the visible text contents of various applications, then uses these contents to generate context queries to information adapters (e.g., a web search) which can return potentially related results [19]. Remem-

brance Agent is an EMACS extension which compares the contents of the open file to other files in the user's workspace, then recommends potentially related files via the interface [50]. Margin Notes applies a similar concept to web pages, where a sidebar recommends local files which share similarity [51]. Implicit Query [13] focuses on email as the application.

TaskPredictor is an extension to the aforementioned TaskTracer that uses file content similarity to identify the user's current task from the set of user specified tasks [52]. This reduces the burden for the user to explicitly label the current task. During the training phase, TaskPredictor uses TaskTracer's monitoring tools to gather the words within recently accessed documents and creates a profile of each task based on word frequency. Once training is complete, gathered words are instead fed into a Bayesian classifier, which identifies the likelihood that those words belong to a different task.

A weakness in content-based approaches is the assumption that content and context are equivalent, which fails in a number of scenarios. Content-similarity may exist between items which do not share a context relationship; for example, responding to an email with a new topic (while retaining the copy of the original message) will generate an email which is largely the same content as the previous email, though different in task. More importantly, content similarity often does not exist where context does; a graph file (containing no text) may be very related to its enclosing report.

Furthermore, content-similarity is static. Two files whose contents are not changed will always share the same content-similarity, while their similarity will constantly be evolving from an activity standpoint. Finally, content-similarity is only as personal as the data on which it is evaluated. Many users may refer to a popular research paper during their work, but the role it plays in their tasks is unique to each user.

Inferred context

The research included in this branch infers task context by observing the user's interaction with their information, and without requiring a semantic understanding of the task within which that interaction takes place nor the information contained within it. Hence, it requires no effort on the part of the user. The theory is that temporally locality, when applied to file use, is a strong indicator of task commonality [43].

One of the core challenges in context identification is in monitoring user behavior. Questions include which types of activity to record and how to best record them. In most cases, some form of instrumentation occurs. Instrumentation is the modification of existing tools such that pertinent events are recorded to some persistent form. Instrumentation is intended to be invisible to the user so as to avoid modifying the behavior which the system is trying to observe. In this section, I describe some systems which trace user activity through some form of instrumentation.

Connections is a context-enhanced file search tool that traces file access events and then identifies inter-file relationships using the frequency of temporally local file accesses [53]. As the user works, files accessed within a temporal window become increasingly contextually related. Figure 2.3 illustrates the architecture. The architecture consists of (1) a tracing module that transparently interposes between applications and the file system, recording all relevant file system operations, (2) a relation graph for maintaining a graph of weighted, directed links among all of the files in the user's system, and (3) a content-based search tool for use in context-enhanced search [53]. Connections's operation relies on two activities: context identification and context-enhanced search.

Context identification converts the gathered traces into the relation graph using a *relation window*, which represents the set of all files which were read within the last n seconds. Conceptually, the relation window is the time interval within which file

operations are likely to be related. When a new write event occurs, a directed link from each of the files in the relation window to the written file is incremented. File accessed together more frequently have higher link weights and are considered more contextually related.

To perform context-enhanced search, Connections runs the user's query through a traditional content-only search tool, generating a ranked list of results. For each result, Connections identifies a subgraph of contextually related files using a modified breadth-first search of the relation graph, limited both by a minimum link strength and a maximum hop distance. It then merges the subgraphs for each result, and applies a graph ranking algorithm (e.g., PageRank) to the merged graph, creating a new ranked list of results that includes files found both by content and by context.

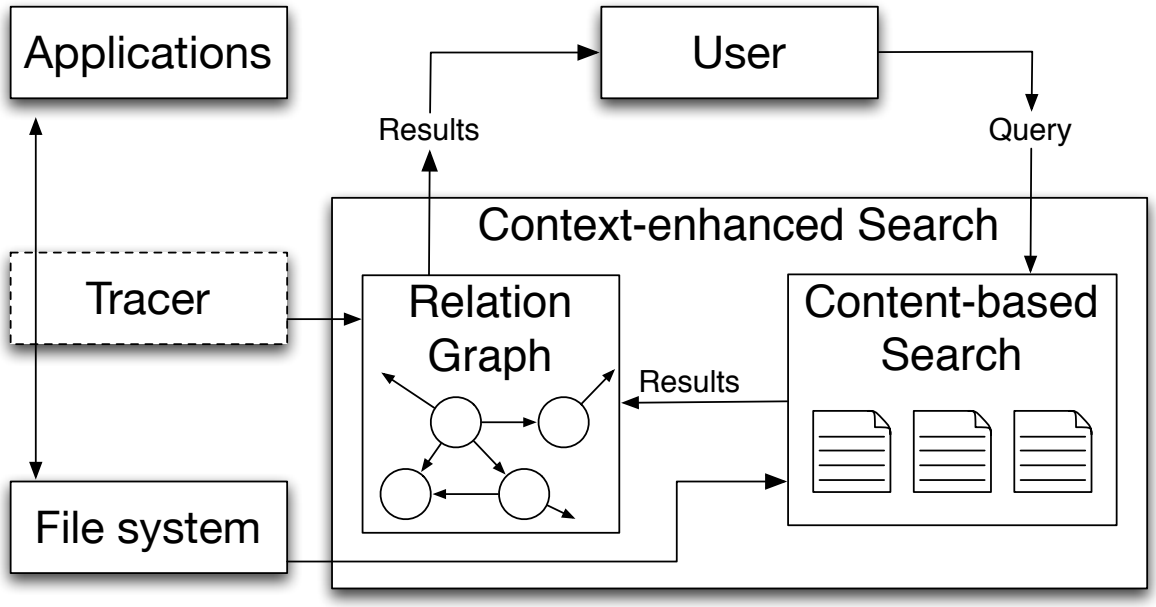


Figure 2.3: Connections architecture.

CAAD is a context-enhanced organizational interface that identifies groups of related files using temporal locality[47]. While Connections's tracing approach is geared toward capturing as much information as possible, CAAD's tracing takes the opposite approach

of trying to limit its tracing to include only files specified as relevant by the applications. While this approach reduces noise, it has the undesirable side-effect of also reducing the amount of information that can be collected, since only supported applications can be traced. CAAD's context identification also differs from Connections by placing each file into a discrete cluster rather than maintaining inter-file relationships. This clustering loses important relationships in the case where a single file is relevant for multiple tasks, making the technique less broadly applicable.

Ivan is a system with a similar approach, deriving relationships from files from events such as (1) files being opened at close points in time, (2) GUI windows displaying files being switched back and forth, and (3) copy-and-paste operations among windows displaying files [45].

SWISH is a system which monitors the user interface for chances in window focus [43]. The theory behind the system is that windows among which the user frequently switches are more likely to share a task relationship. SWISH was therefore designed to recognize patterns in switching behavior.

2.2.4 The great divide

As we have seen, there have been a number of interesting systems developed to include activity in personal informational management. Unfortunately, they remain across a large divide from actual systems due largely to the technical limitations of desktop operating systems, which do not expose the necessary information to make these systems viable. One of the core achievements of this thesis is in connecting these sides of the problem, and, in doing so, we find that novel applications of time and context are possible (especially with the Passages system).

Chapter 3

Confluence: capturing context through file and UI events

Confluence is a tool that automatically identifies contextual relationships among a user's files by analyzing traces of that user's interaction with their system. Building upon the Connections search tool, which extracts relationships from file access events, Confluence also monitors events at the user interface layer. This complementary source of information allows Confluence to isolate recently accessed files that are more likely to be related to the user's current task. By filtering on the current task, Confluence provides a significant improvement in identifying contextually related files over Connections's file-only techniques.

3.1 Introduction

File system data is difficult to organize and search. On the web, hyperlinks between documents describe their contextual relationships, a common method of human recollection [5], and provide the foundation for structural search algorithms like PageRank. Conversely, file system data lacks such local hyperlinks, making it difficult to reason about the contextual relationships among files.

For example, the process of writing a research paper might involve the use of several different files, such as graphs generated by the author's data analysis, source code for software developed as part of the research, and text documents depicting different sections of the paper. While these files are all contextually related, it is not necessarily true that they will share similar content, precluding the use of most existing clustering, search, and automated organization tools.

Consequently, the quality of file system organization and search tools often suffer in comparison to web-based systems. The challenge then is to find accurate techniques for identifying contextual relationships among files.

To address this problem, Soules, et al., created Connections: a contextual file system search tool that uses temporal locality to identify file relationships [53]. Files accessed together frequently are identified as being contextually related, effectively introducing local hyperlinks to file system data. These relationships are then used to augment content search results with contextually related files, similarly to web search. Using these techniques Connections achieves improvements in both recall and precision as compared to content-only retrieval methods. However, despite these improvements, several common file activities can present problems in a purely file based approach: context swaps, application caching and background activity all negatively impact the quality of identified context, reducing the general effectiveness of Connections.

This chapter presents *Confluence*, an enhancement to Connections that addresses these issues by incorporating information from the system's graphical user interface. User interface events expose more insight into the user's task by limiting the set of applications that make up a user's context. By limiting the set of considered files to those accessed by the active applications, Confluence identifies contextual relationships that are more likely to be relevant to the user.

Our user studies with Confluence confirm its benefits. In our controlled study,

Confluence recalls nearly 70% more correct files in total than Connections with at least 20% better precision at all recall levels. In our field study, Confluence recalls over 20% more correct files at 30 results than Connections.

The remainder of the chapter is organized as follows. Section 3.2 describes motivating problems. Section 3.3 explains Confluence's algorithms. Section 4.4 presents our prototype implementation, evaluation technique and results. Section **??** concludes.

3.2 Motivating problems

Applications appear to offer users a way to interact with files directly (e.g., Microsoft word, Photoshop, etc.). However, most applications place abstractions between the user and their files, obfuscating the user's intent from the file system layer and forming the primary source of the file system noise discussed above. The disparity between user-perceived events at the *application layer* and system-perceived events at the *file layer* presents a difficult challenge for systems that rely on noisy system-level tracing alone (e.g., Connections or TaskPredictor) often resulting in inaccuracies. The problems related to this are *origin* and *interaction*, described in section 1.3.2. We specifically address the following problems: **application indirection**, **background applications**, and **hidden actively**.

Application indirection: In response to the user's act of opening a file, an application must request the contents of that file through the file system, generating a series of open and read events. This point signifies that file's transition to the application-layer, after which the application assumes responsibility for moderating further access to the file. While the user may conceptually interact with that file through the application (e.g. reading or editing the document), the application determines if and when to share updates with the filesystem. In the case of a simple text editor such as "Notepad", this file system synchronization occurs only when the user manually saves the file. In

the case of a PDF reader, the file is treated as static content and the program never generates new file operations on the file regardless of how long the user conceptually interacts with it at the application layer. Consequently, it is difficult to reason about the user's interaction with that file by file system accesses alone.

Background applications: Although most users only perform a single task at a time, the set of running applications may be performing any number of tasks in the background on the user's behalf. For example, while the user is editing a text document, other applications may be scanning for viruses, downloading new emails, playing music, etc. Any background file events interleave with the file events of the user's current task, making it difficult to distinguish which are of user origin[1].

Hidden activity: Active user applications can also generate file events without application layer activity. Many applications use a set of configuration and state maintenance files which are invisible or opaque to the user but necessary for the process' execution; frequent accesses by the application to these files create the illusion that these files are part of the user's task when they are conceptually foreign to it. For example, a user opening a set of presentation slides considers only the slides themselves to be part of his context, however, the slide-authoring application may access configuration files, template files, libraries, etc. in the process of displaying the slide deck.

The disconnect between the application and file layers created by these three problems indicates that file access patterns will not always conform to the manner in which users work, and may only glancingly reflect the user's conceptual interaction with their documents. Furthermore, the relationship between the application and file layer varies among applications, making programmatic identification of contextual relationships

[1]For an example of the problem's scale, over the course of 5 minutes of the author editing this document, 1332 read/write events occurred involving 342 files on his machine. Only two documents were being actively used by the author; these documents experienced 7 read/write events between them.

Problem	Summary	Problem space
application indirection	application does not generate file activity even though the user interacts with the file via the application	interaction
background applications	file events are triggered by applications which the user does not interact with	origin
hidden activity	application-specific files that are unrelated to the user's task are accessed by the application with which the user interacts	origin

Table 3.1: Specific problems addressed algorithms

from the file layer both sensitive and brittle. On the other hand, the proliferation of different operating systems, applications, storage types, and document types renders application-specific approaches impractical to implement. This frustrates the Connections algorithms, which desire a clear mapping between user and file system activity.

Users' interaction with their files exhibits locality, and hence exposes implicit information about the relationships among those files; however, the spectrum of applications and file types available to the modern user is ever-changing and increasing, obviating approaches that limit themselves to some subset of them. The only way to capture all file activity is to trace at the lowest layer – the file system itself. Unfortunately, by expanding the amount of information exposed, we dramatically increase the difficulty in identifying meaningful information.

The problem then becomes one of signal vs. noise, with Confluence distinguishing itself from prior art by elevating both to their practical extremes. In maximizing signal and noise, we introduce unique problems that require new methods to solve. Rather than mitigate the noise by promoting our tracing system to a less information-dense layer (e.g. the application), we introduce an additional stream of information from this

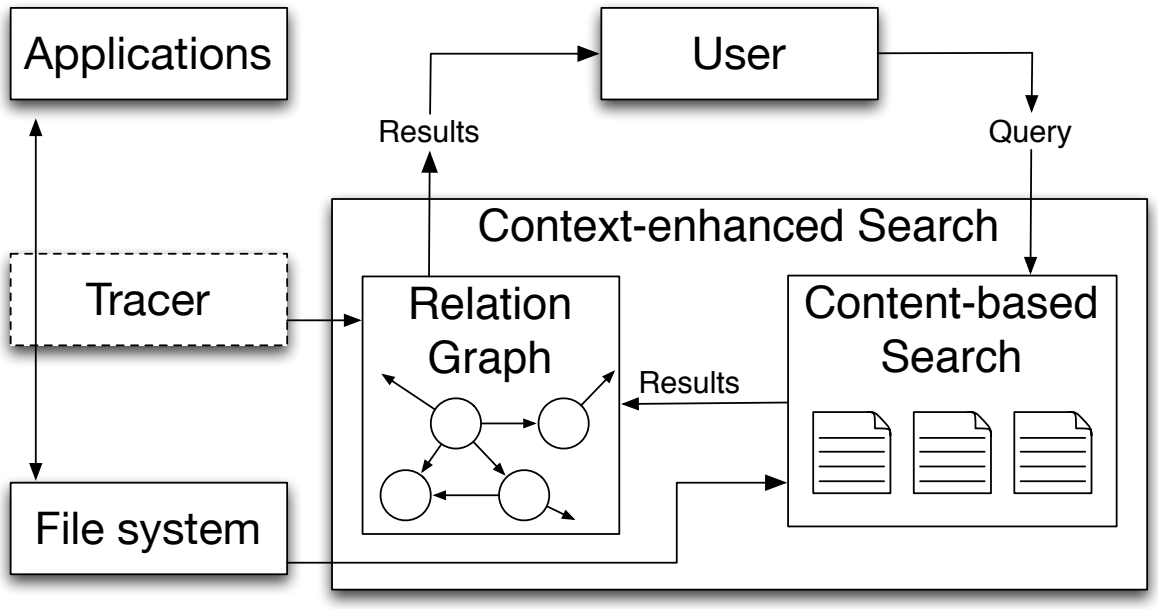


Figure 3.1: Bird's eye view of Connections: Temporal access patterns observed by the file monitor build the relation graph, which is seeded by and augments results produced by a content-based search method.

layer – in the form of user interface events – to complement and inform the analysis of the user's file activity.

3.3 Confluence

Confluence introduces novel algorithms to integrate event streams from two distinct sources: the user interface and the file system. In this integration, Confluence is able to help bridge the disconnect between the application and file layers, enhancing insight into relevant file activity and improving the ability to reason about inner-file relationships. This section overviews the Connections framework, highlighting the relevant components of the system, and describes the changes required to enable Confluence.

3.3.1 Connections overview

Figure 3.1 illustrates the Connections framework. The framework consists of (1) a tracing module that transparently interposes between applications and the file system, recording all relevant file system operations, (2) a relation graph for maintaining a graph of weighted, directed links among all of the files in the user's system, and (3) a content-based search tool for use in context-enhanced search [53]. Connections's operation relies on two activities: context identification and context-enhanced search.

Context identification converts the gathered traces into weighted inter-file relationships. To do so, Connections maintains a *relation window*, which represents the set of all files which were read within the last n seconds. Conceptually, the relation window is the time interval within which file operations are likely to be related. When a new write event occurs, a directed link from each of the files in the relation window to the written file is incremented. File accessed together more frequently have higher link weights and are considered more contextually related[2].

To perform context-enhanced search, Connections first runs the user's query through a traditional content-only search tool which generates a ranked list of results. For each result, Connections identifies a subgraph of contextually related files using a modified breadth-first search of the relation graph that is limited both by a minimum link strength and a maximum hop distance. It then merges the subgraphs for each result, and applies a graph ranking algorithm (e.g., PageRank) to the merged graph, creating a new ranked list of results that includes files found both by content and by context.

[2]Note that we only consider Connections most successful context identification algorithm: read/write [53].

3.3.2 User-interface tracing

The user-interface is an attractive information space because user interface events are, by definition, tightly connected to the user-perceived application layer. By better understanding the user's task as it pertains to the application layer, Confluence can bridge the disconnect between the application and its underlying files.

For the purposes of Confluence, the user-interface consists of a set of *graphical application windows*. As the user interacts with applications, only a single window can be active at any point. Confluence uses window focus events as a cue for the current task. Window focus events are typically caused by an action from the user (e.g., a mouse click in the window region). In effect, the user communicates through the action that there is something on that window that is important for their current task.

During any given time slice, the windows most related to the user's task will be focused more frequently than others, providing insight into their role in the user task [43]. Confluence leverages this knowledge to determine which file events are most relevant to the user's current task.

Confluence captures window focus events at the user-interface level through the use of accessibility support, which is exposed through most mainstream operating system window managers: Windows XP, Aqua (Mac OS X), Gnome and KDE. Accessibility support exposes a programmatic communication channel between third party programs and the UI. This channel enables the third-party program to be notified of interesting, global changes in the UI, such as the most recently focused window, and provides access to information about those events, such as the process which generated the event. Using this channel, Confluence implements a user-interface monitor that transparently records window focus events, requiring no application support.

Confluence merges its stream of user-interface events with the stream of file tracing events within Connections. This interleaving allows Confluence to identify the set of

file accesses that occurred during the focus period for a specific window.

3.3.3 Algorithms

Because context is rooted in the concept of a user task, Confluence ties the current user task to relevant file accesses through the user interface. To do so, it assumes the following task hierarchy: a task is work for a specific goal, such as developing code. Tasks comprise a set of applications, which in turn comprise a set of windows through which users interact with the applications. Each window is associated with a subset of the files accessed while the window was in focus.

We developed four new algorithms in Confluence that extend Connections's original context-identification algorithm using the task hierarchy. They are listed in Table 3.2. We describe the details of each algorithm below.

Algorithm	Approach	Problem space
focused window filtering (FWF) focused task filtering (FTF)	use window focus events to exclude background applications	origin
weight carrying (WC)	replays previous activity for a focused window	interaction
TaskRank (TR)	identifies a file's relevance to a set of files for a given task to address	origin

Table 3.2: Confluence algorithms

Focused window filtering

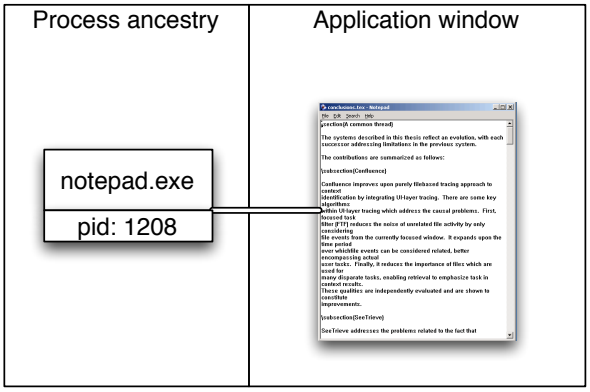
Connections uses a short, fixed-time relation window (e.g., 30 seconds) to limit the growth of false-positive relationships from background applications. Unfortunately, this precludes building relationships between two related files when the respective file operations are separated by a duration longer than the fixed-time interval. Effectively supporting a broad range of task types requires a flexible relation window that more

closely coincides with the user's activity. The Focused Window Filtering (FWF) algorithm assumes that the currently focused window represents the portal through which the user manipulates their files, thus, it ignores file operations that are not generated by the active application, and increases the relevance of those that are, allowing it to expand the relation window.

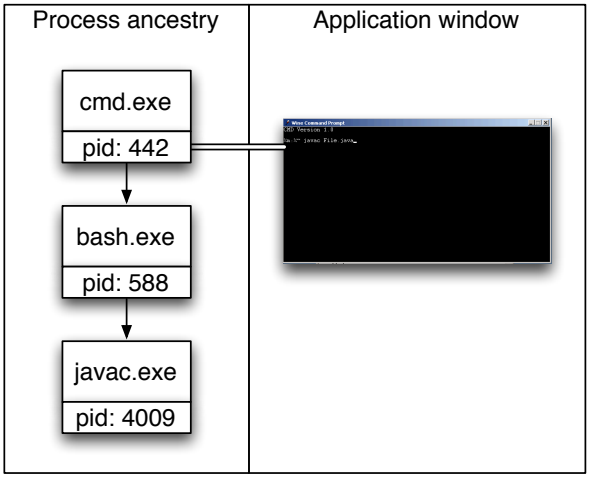
To identify the active application, Confluence maintains a mapping between each window and its process identifier (pid), and considers the pid associated with the active window to be the active application. Only files accessed by the active application, or one of its window-less children, are placed into the relation window. Confluence includes window-less children for cases where a windowed process creates a sub-process to perform work on its behalf. For example, a Java developer working with a Windows command prompt might use JAVAC at the command line to compile a source file; JAVAC would be a sub-process of the command prompt and part of that task, even though it does not have its own graphical window (Figure 3.2).

Because FWF's filtering substantially reduces the volume of file events considered, it can relax the constraints on the relation window. Rather than use a fixed-time relation window that relates all events within that time interval, FWF starts a relation window when an application window gains focus and ends that relation window when the application window loses focus. This matches the relation window's time-span more closely to the user's task, and so is more likely to relate file events that share task commonality. The reduction in file events also provides more flexibility in how the relation window operates. For example, Confluence's relation window connects file read operations to other file read operations, a technique that was plagued by false-positives in Connections [53].

FWF increases the link weights between related files by the inverse of the count of unique files read or written during a relation window. This enhances the strength



(a) Notepad



(b) JAVAC

Figure 3.2: Process ancestry and graphical application windows. In the case of JAVAC, the process has no GUI window. To know that its active, we must identify its nearest parent which has a graphical window.

of the relationships between files during windows where few events occur. This design decision is based on our observation that relation windows in which many file events occur are often the result of large, non-interactive operations, such as the compilation of large source code projects which generate many read or write operations, or revision control commands (e.g., CVS) which scan entire directory structures. Additionally, it seems unlikely that a user would access tens to hundreds of files within a single focus interval and consider them all both important and equally related to each other. The design of these algorithms and heuristics was heavily influenced by observing the traces

of our own activity.

FWF separates read events from write events when calculating graph updates to increase the effect of write events on link weights. Because write events are less frequent, they are more information rich, which is captured by FWF's inverse-file-count weight. Separating reads and writes does not preclude relationships from being built between them, as files are often read before they are written. In practice, the separation only serves to emphasize relationships between written files. Again, this design decision was based on observations on our own data and could be further examined.

The combination of these changes is depicted in Algorithm 1. When FWF sees a newly focused application window, it begins a new relation window that records each file that was read or written by process identifiers that match the currently focused window (as described above). When the window focus changes, the relation window ends, at which point FWF updates the relation graph by incrementing the bi-directional link value between each pair of files read during that interval as well as each pair of files written during that interval.

Algorithm 1 $processNewRelationWindow_{fwf}$

$RW_c \leftarrow$ current relation window
$reads \leftarrow getFilesRead(RW_c)$
$writes \leftarrow getFilesWritten(RW_c)$
for all read file $r_i \in reads$ **do**
  for all read file $r_j \in reads$; $r_i \neq r_j$ **do**
    incrementGraph(r_i, r_j, $\frac{1}{|reads|}$)
    {$|reads|$ represents the number of unique files read}
  end for
end for
for all written file $w_i \in writes$ **do**
  for all written file $w_j \in writes$; $w_i \neq w_j$ **do**
    incrementGraph(w_i, w_j, $\frac{1}{|writes|}$)
  end for
end for

Confluence's FWF algorithm addresses two key problems with Connections: false-

positives generated by the flood of file events, and false-negatives created by the mismatch of the relation window size to the user's perceived context. Despite these contributions, FWF still fails to identify related files whose file events occur across different window focus events (even within a single application). In other words, it cannot link files which were accessed while different windows were in focus. This observation led to the development of the next algorithm: focused task filtering.

Focused task filtering

Focused task filtering (FTF) extends the FWF algorithm to filter file events using the *focused user task* rather than only the currently focused window, broadening the scope of file relationships it can consider. FTF defines the focused user task as the set of recently focused windows among which the user has switched focus as part of their work.

To track the focused user task, FTF maintains a log of relation windows for each window that was focused within the last n seconds[3]. Just as in FWF, when focus changes, the current relation window is ended and the relation graph is updated in two steps (depicted in Algorithm 2). First, FTF updates relationships using only the current window, just as in FWF. Second, it considers each pairing of previous relation window RW_i in the log with the current relation window RW_c. For each pair, it connects the reads in RW_i to the reads in RW_c (and writes-to-writes).

FTF improves on FWF by connecting files across disparate applications, while still reducing the false-positives generated by background applications. By maintaining separate relation windows for each focus period, FTF still prevents a single noisy application (e.g., revision control) from negatively impacting the quality of identified relationships.

[3]Our evaluation considers both 300 and 600 seconds.

Algorithm 2 *processNewRelationWindow*$_{ftf}$: behavior of algorithm on each new relation window

$RW_c \leftarrow$ current relation window
$log \leftarrow$ set of relation windows over last n seconds (not including RW_c)
$reads_c \leftarrow getFilesRead(RW_c)$
$writes_c \leftarrow getFilesWritten(RW_c)$
for all relation window $RW_i \in log$ **do**
  $reads_i \leftarrow getFilesRead(RW_i)$
  $writes_i \leftarrow getFilesWritten(RW_i)$
  for all $r_i \in reads_i$ **do**
    for all $r_j \in reads_{current}$ **do**
      incrementGraph(r_i, r_j, $\frac{1}{|reads_c|+|reads_i|}$)
    end for
  end for{Repeat for writes}
end for

$log \leftarrow log \oplus RW_c$ {Append relation window to list}

Weight carrying

While FWF and FTF address the problem of background activity, they are both still vulnerable to the problem of application indirection. For example, a user may open a PDF file, which issues a read event to the file system and displays the contents to the screen. However, while the user may subsequently focus the window several times to reference the document, no further file events are generated, preventing the system from reasoning about the user's activity. Hence, we lose access to a rich set of interaction which would contribute to a more accurate and comprehensive context.

The weight carrying algorithm (WC), depicted in Algorithm 3, extends the FTF algorithm to address application indirection. For each widget within a window (e.g., tab, text region, etc.), WC caches the last set of file events that occurred while that widget had focus. If that widget is focused again without any new file events being added to the current relation window, WC retrieves the widget's cached events and adds them to the current relation window. This has the effect of creating "fake" file-layer events that attempt to match the application-layer state experienced by the user. This, in

turn, provides Confluence with more information about the user perceived context.

By caching file events at a per-widget granularity, rather than per-window, WC can more accurately model the application-layer context. Many applications present users an interface where windows comprise multiple tabs or edit panels, each of which moderate access to a specific file. Because only the focused widgets are viewed by the user, only the events associated with those widgets should be considered for replay. For example, consider a text editor depicting different files in different tabs. If WC were to reproduce old file events for a tab that never gained focus, it would be obfuscating the file event stream with incorrect data about file use.

Algorithm 3 $processNewRelationWindow_{wc}$

```
lastActive ← last active RW per-widget
RW_c ← current relation window
AW_c ← current application window
for all widget w_i ∈ AW_c do
    accesses ← filesAccessedWhileWidgetFocused(w_i)
    if accesses ≠ φ then
        lastActiveRW[w_i] = accesses
    end if
    if lastActiveRW[w_i] ≠ φ then
        RW_c = RW_c ∪ lastActiveRW[w_i]
    end if
end for
{Call processNewRelationWindow_ftf with augmented RW}
processNewRelationWindow_ftf(RW_c)
```

TaskRank

The *TaskRank* algorithm addresses the final problem of application obfuscation: hidden activity. As described in Section 3.2, many applications employ configuration and state-maintenance files throughout their execution, transparently to the user. The frequency with which these *super-node* files are accessed increase their connectedness in the relation graph as well as the weights of their individual links causing them to

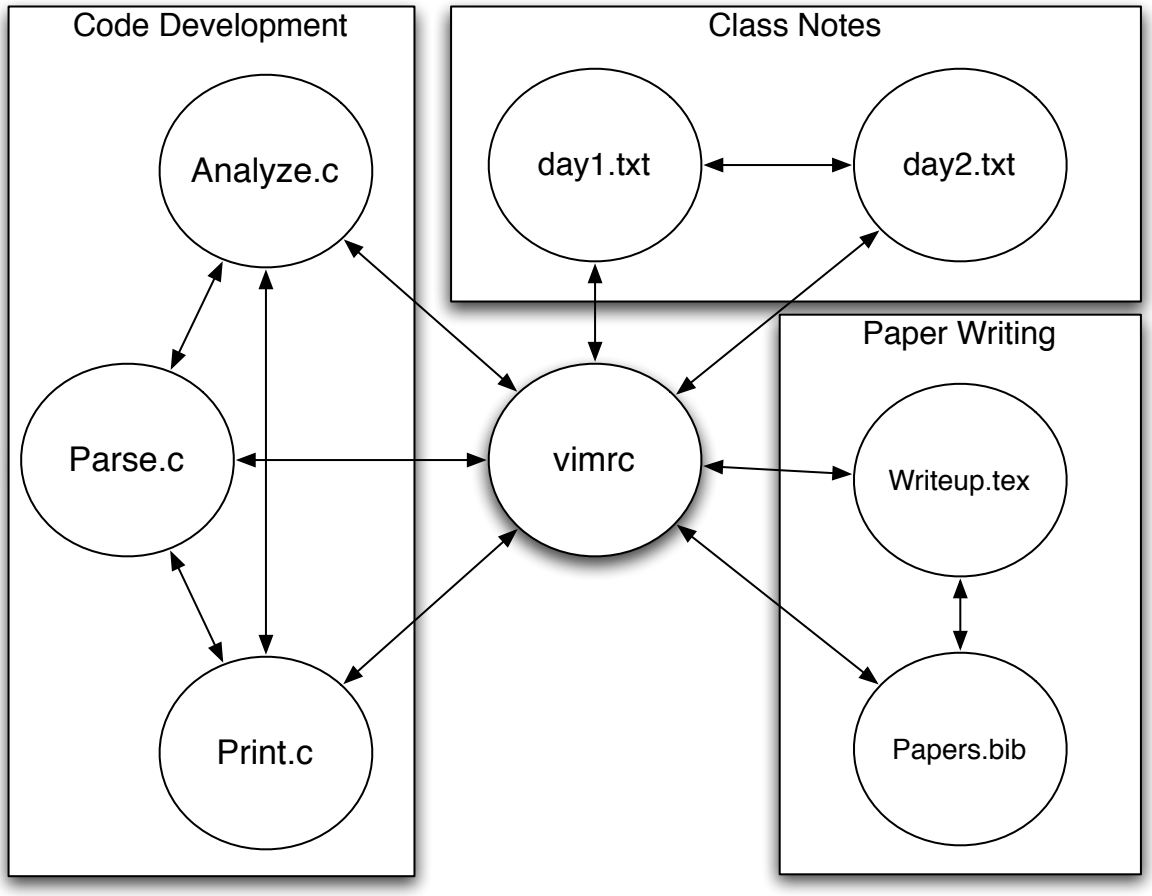


Figure 3.3: *Super-nodes:* In this subgraph of a user's relation graph, *vimrc* – a configuration frequently accessed by *vim*– features incoming weight from each task using *vim*, despite being conceptually unrelated to the tasks.

falsely appear related to the user's task.

Manifested on the relation graph, super-nodes feature a disproportionately high number of links and tend to bridge otherwise distinct tasks, as illustrated in Figure 3.3. In this respect, context based search diverges from web search methods, such as PageRank, which associate a high degree of incoming links with the authority or credibility of a page. The task based nature of context means that, generally, quite the opposite is true; the more tasks with which a file shares strong links, the less likely it is that file has a meaningful role within any particular task.

TaskRank introduces a value, S, that represents the exclusivity of the relationship between a given file and a given task-based file set. This value can be applied a priori to the relation graph, for example while running a context-enhanced search, to identify and reduce the impact of super-nodes. Equation 3.1 defines the *TaskRank* for a file f to a task-set T. Let F be the set of files that are linked to file f. The top half of Equation 3.1 calculates the sum of the link weights from f to each file in $T \cap F$. The bottom half of Equation 3.1 calculates the total weight of all links from f.

$$TR(f,T) = \left(\frac{\sum_{f_i \in F \cap T : f \neq f_i} linkValue(f, f_i)}{\sum_{f_i \in F} linkValue(f, f_i)} \right)^2 \tag{3.1}$$

Conceptually, *TaskRank* represents the amount of a file's total link weight that is part of a given file set. A *TaskRank* value close to 1 indicates a file's relationship to a task is close to exclusive, while a value close to 0 indicates a file is related to many other file sets. For example, if file B has $TR = 0.9$ for the set of files to which file A is connected, it is likely they are part of a similar task. Conversely, if file C has $TR = 0.1$ with respect to the neighbors of file A, it is unlikely to be part of a common task – even if the link value between them is high.

While not a specific application of user-interface information, *TaskRank* presents an additional way to deal with causal dissonance. Confluence applies *TaskRank* to Connections's context-enhanced search to promote files which share similar file relationships. We consider one such application in our evaluation, as described in Section 4.4.

One might believe that *TaskRank* excludes files from being part of multiple tasks, however, we see in practice that cases where a user's file is part of multiple tasks yield a high TR value when compared to application configuration files, which are part of every task that involves that application. Furthermore, TR is applied to promote the position of files within a result list, rather than as a threshold which removes low scoring files from consideration, affecting precision, not recall.

3.4 Evaluation

The goal of Confluence is to accurately identify contextual inter-file relationships that can be used by context-aware applications. Correspondingly, our evaluation aims to identify the extent to which context — as represented by the relation graphs built by various algorithms — reflects the user's perceived conceptual relationships. We conducted two independent user studies to evaluate Confluence: one controlled study where users worked on predetermined tasks, and a longitudinal field study.

Data collection for each experiment used Confluence to trace events. Confluence includes a file system event monitor and a user-interface event monitor that record the events listed in Table 3.3. Confluence also traces all process creation and exit events, allowing it to (1) maintain necessary mappings between a process's file descriptors and the files they represent, and (2) maintain the process hierarchy required for FWF and FTF as described in Section 3.3. From a performance standpoint, Confluence's tracing does not contribute a noticeable or significant performance penalty on modern personal computers.

Processes	Create, Close, Fork, Exec
Files	Read, Write, Open, Close
UI	Focus change for window and widget

Table 3.3: Traced events

Our evaluation compares four different algorithms for generating relation graphs: Connections's read/write algorithm (as described in [53]) and Confluences three algorithms described in Section 3.3 (FWF, FTF, and WC)[4]. For the FTF and WC algorithms we compare relation windows sizes of 5 minutes and 10 minutes.

Identifying related files to a given file requires a graph search. We implement graph

[4]We considered including CAAD's context identification algorithm, however, without a better description of how they identified and limited files that were "in use", we were unable to generate a meaningful relation graph using their techniques.

search using two algorithms. When searching Connections's relation graph, we use its Basic-BFS algorithm (also described in [53]). When searching Confluence's relation graphs, we retrieve the set of files F that are linked with the seed file f_x and assign each $f_i \in F$ a ranking-weight equal to the product of the link weight w_{xi} and the *TaskRank* value $TR(f_i, F)$.

3.4.1 Experiment I

For our first evaluation, we used a controlled user evaluation in which users accomplished a set of computer-mediated tasks involving different forms of file interaction while our system traced their activity within the UI and filesystem. Rather than have a few users work for long periods of time, our study involved a larger number of users working for shorter periods of time on a shared computer. Their activity was traced using our file tracer, with records placed within the same database, simulating a single user working over a longer period of time.

In this evaluation, we had users complete one or both of two predetermined tasks involving computer use. The users completed the tasks in succession, using a laptop we provided.

Task 1 was the creation of a conference trip report using a wiki, a web-based collaboration tool interfaced through a standard web browser, installed on a separate machine. The user was asked to create a wiki page briefly describing three papers from a fictitious conference. The user was instructed to choose three papers at random from a pre-generated corpus of conference paper files[5], skim each paper, write a brief (1-2 lines) summary of the paper on the wiki page, and upload the paper to the wiki. Once the user selected a PDF it was removed from the corpus to prevent overlap with other users.

[5]Papers were selected from the ACM Digital Library.

Task	Description	Files
1	Read 3 ACM papers and write a summary/review page describing them	3 PDFs, 1 summary page
2	Research 3 items from provided topic, download images of these items, upload these images to Flickr, tag and add metadata to them from research	3 images, 3 image sources (e.g., Wikipedia), 1 photo album page

Table 3.4: Task descriptions for Experiment I

Task 2 was the creation of an online photo album. The user started by creating a photo album using a photo album website installed on a separate machine. The user was given a topic (e.g., marine animals) and asked to identify three items within that topic (e.g., dolphins, manatees, and orcas). For each item, the user was asked to acquire an image of that item online, download it to their machine, and then upload it to the photo album. The user was then asked to provide a brief description of that item as researched online (e.g., through Wikipedia) and place that description within the "description" category of the photo on the photo site.

The controlled nature of the study meant that the files used with each task were well defined. Specifically, each *pdf* file accessed was considered part of the review task, while each *jpg, gif, png, etc* file were considered part of the photo album task.

This experiment was originally designed to evaluate the SeeTrieve system described in Chapter 4. However, we were able to use the traces generated during the study to compare Confluence's contextual relationships to those formed by Connections. We acknowledge that this experiment is limited to a set of handpicked applications and file types, and hence at odds with the spirit of generalizability of Confluence. However, it does enable us to isolate the influence of our new algorithms in a particular context, as well as get an empirical feel for the prominence of noise from background activity and

TaskRank's ability to mitigate it.

In total, we recruited *16* volunteers for our study. Volunteers were recruited by a listserv email within HP and paid by a gift certificate usable internally for HP products. 13 accomplished the tasks. 53 files were used in $task_1$ and 37 were used in $task_2$ (roughly, 3 files were used for each task, although some volunteers used more files).

Noise

Despite being a controlled experiment, the percentage of relevant files within the entire set of files accessed during the user's task was extremely low. The total number of unique files accessed during the trace was 2116, meaning under 4% of files accessed were actually interacted with by the user. Furthermore, file accesses were filtered to include only those contained within the user's home directory or any subdirectory within it; without such filtering, file accesses from system directories would have likely substantially increased the access count. This demonstrates the overwhelming volume of information exposed at the file layer, and emphasizes the need for systems like Confluence to identify the most relevant data.

Relation graph

For each of the algorithms, FWF, FTF (300 and 600 seconds), WC (300 and 600 seconds), and Connections, we built $n+1$ relation graphs: one from each user's individual trace and an additional graph using all users' combined traces. To create the combined trace, we modified the time stamp on each trace such that its initial event occurred immediately after the last event from the previous trace.

For each file in a task, we executed a search on each method's relation graph to determine how many of the other files within the task were within its result pool[6].

[6]As mentioned above, we rank Connections's results using Basic-BFS and all of Confluence's algorithms using TaskRank.

We considered two metrics: total recall and precision at 11 recall levels (i.e., a recall/precision curve). Total recall, the number of correct results found by a given scheme divided by the total number of correct results identified by all schemes, shows the maximum potential of each scheme to identify the related files. The recall/precision curve, the average precision of each scheme at 0%, 10%, ..., 100% recall, illustrates a schemes ability to identify related files quickly. Note that the value at 0% is the maximum precision from 0% to 9%, and so on. If a scheme cannot reach a certain recall level, its precision at that level is considered 0%.

Table 3.5 lists the total recall values for each method within each task. FTF and WC significantly outperform Connections due to Confluence's filtering, which enables it to substantially expand its relation window. The increase in recall from a 300 second to a 600 second relation window for both FTF and WC indicates that user tasks span large periods of time, further confirming our claim. As expected, FWF underperforms due to its limited, single-application scope.

Figure 3.4 illustrates the recall/precision curve for the six schemes under comparison. There are three interesting points to draw from this graph. First, while WC is able to recall more correct results than other schemes, it does so at a cost of precision, indicating that weight carrying introduces more noise than signal. Second, the higher precision at lower recall levels indicates that FTF does a good job of reducing background application noise. Third, Connections significantly underperforms the other schemes, even FWF, despite higher total recall. Because Connections's algorithms rely on link strength to distinguish noisy links, it is unable to capture relationships from single instance tasks such as our controlled study, another drawback of Connections's approach.

Method	$task_1$	$task_2$	Combined
FTF (300 seconds)	0.542	0.682	0.574
FTF (600 seconds)	0.694	0.955	0.755
WC (300 seconds)	0.569	0.705	0.601
WC (600 seconds)	0.701	0.955	0.761
FWF	0.000	0.227	0.053
Connections	0.000	0.307	0.072

Table 3.5: Recall performance for Experiment I

TaskRank

Because *TaskRank* minimizes the impact of super-nodes, we expect it to exhibit two properties: (1) improved precision, and (2) increased effectiveness as the number of tasks increase. We evaluated *TaskRank*'s improved precision by evaluating FTF-600 with and without *TaskRank*. We evaluated *TaskRank*'s increased effectiveness with more tasks by comparing FTF-600 created from a single user's trace to FTF-600 created from all users' traces[7].

Figure 3.5 illustrates the recall/precision graph for the four setups that represent the cross-product of our two evaluations. As expected, *TaskRank* improves the average precision of results and this improvement grows as the number of witnessed tasks increases. The fact that the worst performance was observed when considering all the users' data without *TaskRank* emphasizes the need for its noise reducing abilities.

3.4.2 Experiment II

Our second experiment evaluated the effectiveness of Confluence over a longer period and on live systems. Users installed and ran the Confluence tracing software on their primary-use computers for 3-6 weeks, which maintained the relation graphs for five schemes as they worked (we omitted FWF due to poor performance observed in earlier

[7]FTF-600 was chosen because the impact of *TaskRank* was most pronounced on this method, although the results were similar for other methods.

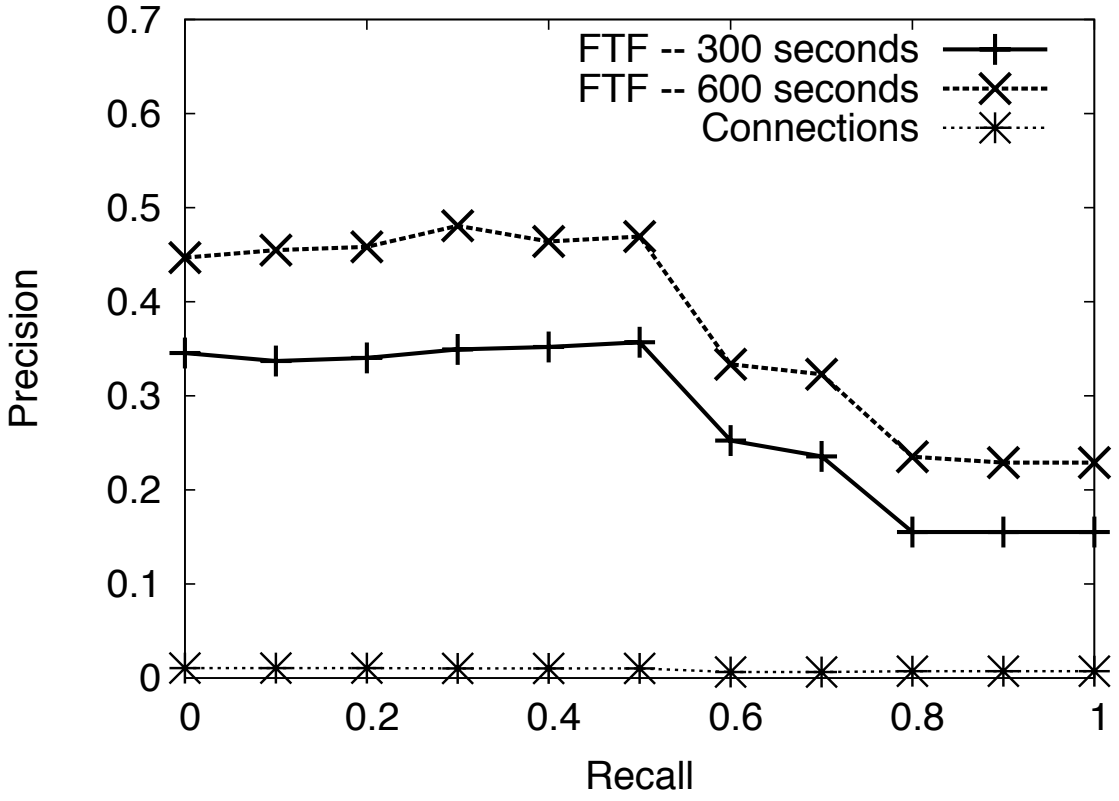


Figure 3.4: Recall/Precision performance for Experiment I

experiments). Because we did not control any of the user's tasks, and were not exposed to the user's data, it was impossible for us to know the complete set of files that made up any single user task, thus making the evaluation technique used in our controlled study impossible.

Instead, each user identified from memory a set of 5-10 disjoint tasks with which they were engaged at some point during the tracing period. "Task" was defined as any goal that required at least two files to accomplish. "Disjoint" was defined to mean that the tasks have minimal overlap of files with other tasks. For example, two separate homework tasks could refer to the same document containing needed equations, meaning they overlap. For each identified task, the user selected a *seed file* that was used as part of that task.

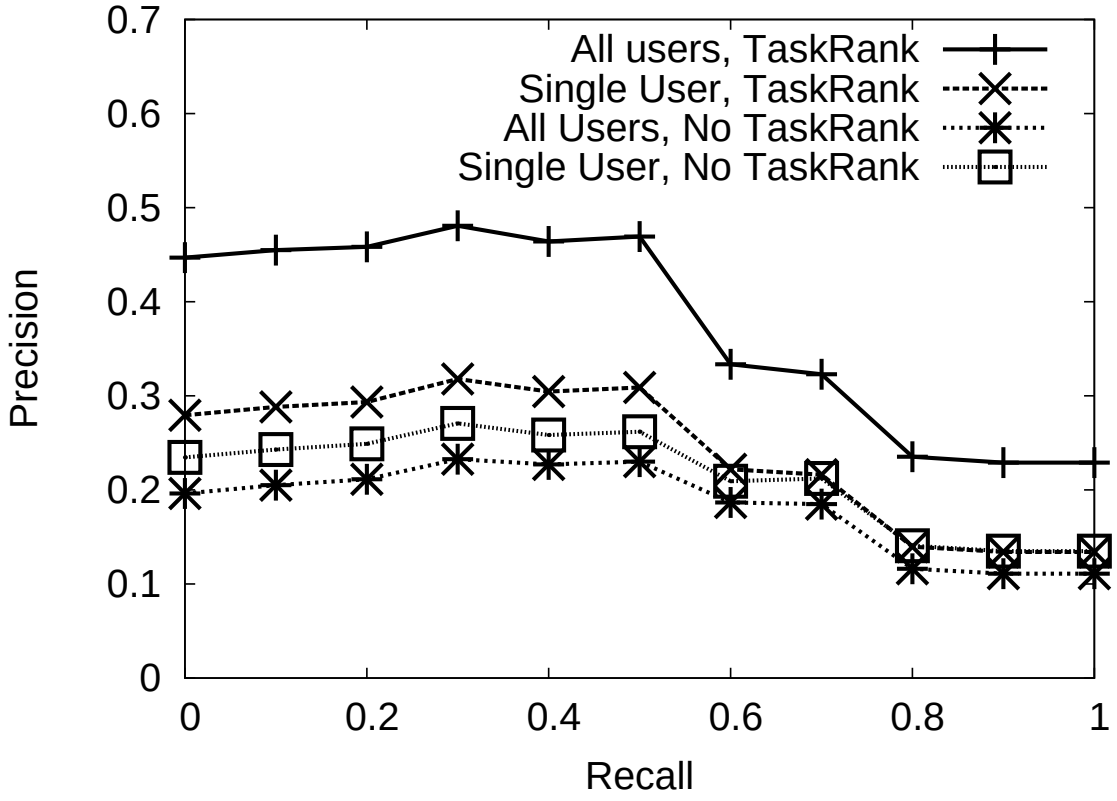


Figure 3.5: Effects of task rank and data size for FTF-600; Experiment I

Confluence performed a search from the seed file in each relation graph, creating a list of related files for each scheme. Rather than having users rank each exhaustive list separately, a time consuming and frustrating task that can quickly introduce user fatigue, we pooled the related files for each seed file into a single *merged list* of unique results, sorted alphabetically. To increase coverage of potentially related files, we also merged results from a directory search algorithm that produced a list of all files that existed at some point within the same directory as the seed file, under the premise that user's directory organization of their files at least partially reflects the commonality of those files. To further reduce user fatigue, we capped the size of the merged list to 100 items, removing the lowest ranked item from each list of related files in a round-robin fashion until the limit was reached.

Method	5	10	15	20	25	30
FTF (300)	0.17	0.23	0.30	0.37	0.42	0.46
FTF (600)	0.17	0.22	0.29	0.33	0.40	0.42
WC (300)	0.12	0.23	0.30	0.34	0.38	0.42
WC (600)	0.12	0.21	0.31	0.34	0.39	0.42
Standard	0.11	0.13	0.16	0.17	0.17	0.17

Table 3.6: Average recall by Result Size: Shaded cells indicate a statistically significant difference with Connections, derived from one-sided t-test ($df = 71$). Dark gray is significant with $P < 0.01$, light gray is $P < 0.05$.

Users were presented with the merged list for each seed file, and asked to rate each listed file on a 0-3 Likert scale, with 3 indicating a strong relationship to the seed file and 0 indicating no relationship. Because of ambiguity in defining a file that is "partially" related to another, we conservatively treated any file which did not receive a rank of 3 as unrelated. As tasks can vary widely in number of files, we observed a variety of pool sizes. Using these ranked pools, we evaluated each algorithm by total recall, or the percentage of the algorithm's results which had a rank of 3.

Findings

Our field study involved *6* volunteers, a combination of university graduate students and industrial researchers, who were traced for a period of 3-6 weeks. During the evaluation, the users' file selections identified **36** seed files.

Table 3.6 depicts each algorithm's recall value at different related files list sizes averaged over the **36** seed files. For a result size of 5, the different algorithms performed similarly. As result size increases, the improvement of the Confluence algorithms over the pure file-based approach grows substantial. This improvement is significant based on a one-sided t-test using 71 degrees of freedom ($P < 0.05$ for all result sizes 15 or greater; $P < 0.01$ for all result sizes 25 and greater).

Other findings are as follows:

- There was no statistical difference in recall between the *FTF* and *WC* algorithms. However, as experienced in the controlled study, *WC* did find unique, accurate file relationships, and performs better at higher result sizes (50-100).
- In cases where users remember some but not all files from a task, Confluence would be an effective retrieval tool. This case presented in the experiments.
- Filesystem noise is significant and damages context building, and any system attempting to support file-based context on the desktop must contend with it. While Connections's file-based approach captured the same file activity as Confluence, by mitigating this noise, Confluence's UI-aware methods enable relevant relationships to be more apparent, improving recall at lower result cutoffs.
- The increased relation window duration and file operation flexibility enabled by the filtering is advantageous.
- Noise from hidden file activity grows with more data, and is mitigated by *TaskRank*.
- FTF's task-recall changes little between 5 and 10 minute durations. While user tasks often take longer than 10 minutes, the majority of events between related files occur within 5 minutes of each other.
- For 28 out of the 36 queries, FTF was successful in producing highly related files (90% success rate). Furthermore, only one user experienced more than a single failed query (2 failed queries), meaning FTF was consistently successful for all users.

3.5 Limitations

We limited consideration to window focus events in this work despite the fact that many more events are available through the UI event stream. It may be beneficial to

trace other events in the UI. Future work could include other UI layer information such as widget scroll events; text selection, copying, and pasting; more finely grained focus events (e.g., a user may have a tabbed browser window where different tabs correspond to different tabs); and window visibility (e.g., an unfocused window with large portions of its content remaining visible).

We assumed that the focused window represents the user's current focus. This is not always true; the modern UI allows for windows specifically to allow users to view multiple windows of information simultaneously. We have already considered examples in which this may be problematic. For example, the lengthy amount of time required for compiling many source files sometimes encourages users to switch to other, unrelated windows while waiting for the process to complete. This means that file events pertaining to the window that lost focus no longer have the ability to be related. Though flawed, our focused window assumption simplified our problem and produced useful results. It would be interesting to expand our work to include multiple windows which may have the user's focus (but not necessarily UI focus).

We did not observe a statistically significant difference between WC and FTF, although they did have differences. This indicates that the distinction between these approaches could be beneficial. Though this work did not directly follow this up, the subsequent systems SeeTrieve and Passages addressed similar ideas which were proven to be fruitful. WC in its implementation, though an idea along the right track, was too blunt to be more effective.

3.6 Concluding remarks

This research presents a compelling case for UI layer events as a medium through which user task can be understood. The incorporation of window focus data improves the quality of task-based context. Importantly, UI layer events are easy to record, place

little burden on the system, require no direct application support, and closely coincide with direct user action. This work offers a more general contribution in that it is further evidence that file relationships can be identified through similarities in their access patterns. Contextual relationships should be viewed as a promising dimension along which a user's files can be understood.

Chapter 4

SeeTrieve: building information context from what the user sees

SeeTrieve was originally designed to overcome the shortcomings of the Confluence system. Though Confluence supports task-based retrieval by identifying inner-file relationships, the overarching theme is to identify relationships among information. Hence, the SeeTrieve system adopts a more generalized approach in which text which the user views in the interface is associated with local files which are used at proximal times to the viewed text. This chapter describes the SeeTrieve design, evaluation, and results.

4.1 Introduction

The definition of a user's file is growing increasingly abstract. With the maturing of distributed systems, especially the web, the modern user's document activity occurs over a broader range of heterogeneous applications and locations. In the recent past, the creation of a spreadsheet would be a local task, involving the use of an application that resided on the author's computer. Today, a user can create and edit a spreadsheet associated with an online office suite like *Google Office*. While the interaction paradigm

for the user remains largely unchanged (e.g., entering data into cells), the way in which that interaction manifests on a file has changed dramatically.

Problematically, not only does a user's file activity not need to take place on a local file, but it does not need to take place on a file within a domain of that user's control. For example, there is no clear way for a generic local search tool to be able to index and retrieve an online spreadsheet. As an exception, a tool like *Google Desktop* is able to retrieve a user's online documents but only because it (a) has *a priori* knowledge of what a Google Spreadsheet is, and (b) comes from the same proprietary source and hence resides within the same ownership domain. If a new website offers a competing online office suite, existing tools would have to be (a) retooled with an interface to the new application, and (b) given access into the same ownership domain. Clearly, task management and search tools are faced with a moving target.

One problem with Confluence is that it assumes a concrete definition of a file, like a text document, email, or web page. Unfortunately, this requires some interface to applications in order to obtain this information. For example, email clients store emails in different ways, making it impossible to simply scan a directory for the contents of emails. Furthermore, applications often enable users to interact with content that has no palpable, indexable manifestation. For example, a user might interact with a dynamic web page whose content is available only in that instance and cannot be cached by a local search tool for later retrieval.

Traditional content-based schemes for classification and retrieval underperform on personal data sets because they fail to map the user's context to the file's contents. On the one hand, they lose information presented to the user through other sources. For example, a user may be referring to information on an email while editing a related document; in this case, a content-based scheme has no way to include that contextual information when indexing the document. On the other hand, they integrate additional

information that the user may be unaware of. For example, a user may open a PDF and look at only the first page; however, a content-based scheme will index all of the pages, which may include terms unknown to the user.

SeeTrieve addresses the disconnect between user-perceived context and file contents. Instead of indexing file contents, SeeTrieve captures and indexes text displayed at the user-interface level by applications. Using temporal locality, it creates a mapping between text snippets and files used while the snippets were displayed. This extends the traditional two-level mapping of terms to documents to a three-level mapping of terms to snippets to documents. Just as in a two-level index, SeeTrieve can use its three-level index to both classify documents, finding relevant terms from related text snippets, as well as retrieve documents, searching the index of text snippets and then following them to relevant documents.

Conceptually, the SeeTrieve's three-level index performs *task-based* classification and retrieval by matching the contents of displayed text during a given task to the set of files used for that task. For example, a user who forgets the name of a file attached to a past email might remember the contents of the email. Issuing a query to SeeTrieve that matches the contents of the email would return the attached file.

To showcase the value of SeeTrieve's three-level index for retrieval and classification, we implement two applications: document retrieval and context tagging respectively. Our user study with document retrieval shows that SeeTrieve recalls 70% more data on task-based retrieval without a loss of precision, and recalls 15% more data on known-item retrieval with only a slight drop in precision. Our user study with context tagging shows that SeeTrieve's classification is considered accurate by users, even when documents contain no indexable data (e.g., images).

4.2 Motivation

Despite the continued abstraction of the user's document space, users' interactions with their documents have changed comparatively little. Users populate a spreadsheet in the same manner whether using Excel (local) or Google Spreadsheets (remote). Writing an email does not functionally change, regardless of whether the user does so with Outlook, Pine, or Yahoo's web mail. This signifies a trend of divergence between the user's activity in the *user interface layer* and its complementary activity in the *file system layer*.

This divergence makes the user interface layer an attractive space to capture activity context because it (a) is very tightly coupled with user interaction, (b) involves activity which is less sensitive to change (e.g., reading, typing), and (c) exposes the contents of the objects with which a user interacts, even when those objects have no "indexable" form (e.g., they are not local, they are application specific files, etc.).

Let us revisit the five main problems of the divergence between the application layer and the file layer (described in Section 1.3), within the context of SeeTrieve. SeeTrieve's combination of user interface and file layer information mitigates or minimizes each of the problems. *Location* and *composition* are dealt with by capturing the contents of remote files or files with proprietary formats at the user interface layer. *Presentation* issues are resolved by the fact that the user-interface layer displays only what the application intends the user to see (e.g., the currently read page of a large document). *Interaction* issues are, by definition, best handled at the user interface layer, where information about what is and is not visible is managed. *Temporality* issues are resolved by SeeTrieve's algorithms that combine the timing of user interface events with the timing of file events to create a mapping from text snippets to files weighted by the level of user interaction.

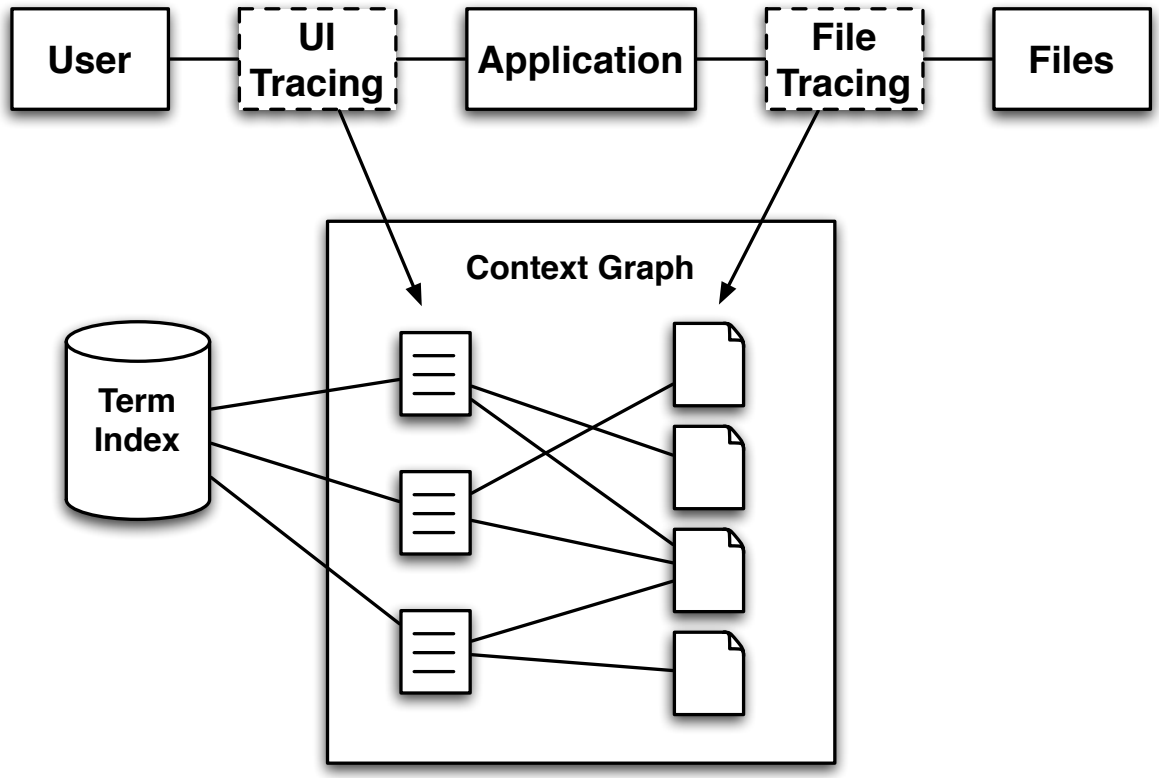


Figure 4.1: SeeTrieve architecture.

4.3 *SeeTrieve*

Figure 4.1 illustrates the design of SeeTrieve. As the user interacts with applications and data, SeeTrieve collects data about their context, capturing visible text into *snippets* and tracing all file activity. Using these traces, SeeTrieve creates a bipartite *context graph* that maps between snippets and files. Finally, as examples of the utility of the context graph, SeeTrieve provides two applications: a document retrieval tool that combines a traditional search index on the snippet contents with the context graph, and a context tagging tool that identifies relevant terms for files based on their snippets. This section discusses these two components (data collection, context graph) and the example applications in more detail.

4.3.1 Data collection

Context-aware systems must be able to understand the user's behavior, independent of which applications and file formats they use. SeeTrieve solves this problem by tracing both user interface and file system events at the operating system layer.

These traces capture four pieces of information that SeeTrieve requires: text snippets of what the user sees, the times at which these snippets become visible to the user, the duration over which these snippets are visible, and the times at which files are accessed.

SeeTrieve acquires text snippets through the accessibility functionality, which has been historically used to enable third party applications that programmatically interact with the user interface to support impaired users. Accessibility data is exposed by most mainstream operating system graphical interfaces, including Windows XP, Mac OS X's Aqua, and Gnome. Accessibility support enables custom programs to query arbitrary applications for information about their UI state, such as which tab or pane is currently focused, and the contents of a text area. While accessibility information can be designed by an application's developers, the use of system components in UI construction means that much of this information is already provided.

SeeTrieve traces the activation and minimization of application windows, allowing it to infer when windows go in and out of visibility and the duration over which they remain visible. Whenever a window changes visibility, SeeTrieve does a full capture of all visible text on the currently active window into a snippet and inserts that snippet into the stream of trace events. SeeTrieve also does periodic captures once every 3 seconds to handle cases where focus doesn't change, but the visible text does.

SeeTrieve traces file READ and WRITE operations to identify which files are accessed when. This trace of file system events is later merged with the trace of user interface events when creating the context graph.

4.3.2 Context graph

The relationship between snippets and files is represented by a bipartite *context graph*, with links between nodes indicating the strength of the contextual relationship between them. Creating the context graph requires two steps. The first is to merge similar snippets together, the second is to pair merged snippets with their related files.

Merging snippets

User activity often involves switching among multiple applications or windows. Because SeeTrieve treats every focus event as a new source of text, such activity can generate many snippets of identical text that originate from the same conceptual document (e.g., the same web page). Most classification and retrieval techniques rely on a discriminating value of terms in the corpus (often inverse document frequency). If a term appears frequently within a snippet while relatively infrequently in the rest of the corpus, it is considered informative. Consequently, populating the corpus with many duplicate snippets reduces SeeTrieve's effectiveness at classifying and retrieving documents. Hence, we implemented a document similarity technique to merge similar and identical snippets to substantially reduce this effect.

By merging similar snippets, and not just identical snippets, SeeTrieve can deal with slight changes in visible text (e.g., status bar updates, open menus) while still identifying completely separate snippets (e.g., the next page in a PDF, a new web page). SeeTrieve identifies similar snippets using the *Max Hash* algorithm [16]. *Max Hash* uses landmark chunking (implemented with Rabin fingerprinting [46]) to break snippets into variable sized chunks. Landmark chunking has the advantage that, because the chunk boundaries are chosen based on content rather than a fixed size window, small changes to the file will only change a small number of the chunks. Each chunk is then hashed using the *MD5* function, and the hashes are sorted numerically. If the top n hashes

of two snippets' chunks match, then it is very likely that the snippets are similar[1]. In practice, we treat any two snippets that share more than half of their hashes as identical. Since snippet size is governed by the amount of text which can appear on a screen, the number of hashes for a snippet is small and sharing half of these hashes indicates with highly probability that the two snippets are very similar.

We chose *Max Hash* as our similarity metric because it is (a) robust to small changes in content and (b) efficient in performance and space. To find if a snippet has an existing similar snippet, SeeTrieve maintains two hashtables. The first contains *Max Hash* values as keys and snippets containing that hash within their top n hashes as values. The second is the reverse: snippets are keys and their top n hashes are values. When SeeTrieve witnesses a new snippet, it is chunked and hashed. For each of the top n hashes, SeeTrieve queries the hashtable for any snippets that contain the hash. It then finds the top n hashes for each matching snippet. If at least $\frac{n}{2}$ of the new snippet's top n hash values match an existing set of hash values, the two snippets are considered similar. This process requires only n lookups to find a similar file and the list of hashes for each file is n 32-bit values, thus both the computational and storage requirements are small.

Pairing snippets to files

The link weight between a snippet and a file node is increased when snippet S is seen in close temporal proximity to an event on file F. SeeTrieve captures this proximity through a *context interval*, a time period during which witnessed snippets are considered to be related to that file. A context interval of n seconds means that any snippet S witnessed less than $\frac{n}{2}$ seconds before or after an event for file F is related to F. Thus, snippets and files that are more frequently proximal will, generally, have higher relative

[1]The higher n, the more similar the snippets must be.

link weights between them.

SeeTrieve strengthens links using two factors: *duration* and *temporal proximity*. Duration measures the length of time over which a snippet was visible. Intuitively, this captures the relative importance of the data contained within it. Let S_{start} be the point at which snippet S is seen that is not similar to the previous snippet in the trace, S_{end} be the point at which a new snippet that is not similar to S is seen[2], t_F be the time at which file event F occurs, and ci be the duration of the context interval. Then, Equation 4.1 defines the duration value for a snippet S and file F.

$$dur(S, F) = \frac{min(t_F + \frac{ci}{2}, S_{end}) - max(t_F - \frac{ci}{2}, S_{start})}{ci} \tag{4.1}$$

Temporal proximity measures the temporal distance between the snippet and a file event. The closer in time a snippet appears to a file event, the more likely it is to be related to the file event. Weighting by temporal proximity relates events over a longer period of time without introducing too much noise (e.g., an infinite context interval equally relates all snippets to all files). Then, Equation 4.2 defines the temporal proximity weight between snippet S and file F.

$$prox(S, F) = \begin{cases} S_{start} < t_F < S_{end} & 1 \\ o.w. & 1 - \frac{\min(|t_F - S_{start}|, |t_F - S_{end}|, \frac{ci}{2})}{\frac{ci}{2}} \end{cases} \tag{4.2}$$

When snippet S is visible at some point within the context interval of file F, SeeTrieve increases the value of the link between them by the product of duration

[2]The definitions of S_{start} and S_{end} merge sequences of similar snippets into a single snippet for the purposes of measuring visibility time. Due to polling, multiple snippets may correspond to a single window that has maintained focus.

and temporal proximity.

4.3.3 Application 1: document retrieval

SeeTrieve implements document retrieval by combining a traditional content index[3] built over the snippet contents (shown as the term index in Figure 4.1) with the context graph. It maintains the content index by adding new snippets (i.e., snippets with no similar existing snippets) as they are seen.

To retrieve a document given a user query, SeeTrieve first passes the query to the content index to identify relevant snippets and then uses the context graph to identify related documents. Specifically, the content index returns a pool P that contains a list of $\langle S_i, V_i \rangle$ tuples where S_i is a snippet and V_i is its corresponding relevance score. SeeTrieve then does a search on the context graph to identify R, the set of files most related to P.

R starts as an empty result pool to be composed of 2-tuples containing a file and its relevance score. For each snippet $\langle S_i, V_i \rangle \in P$, SeeTrieve retrieves each link to a local document $\langle F_j, L_j \rangle$ where F_j is the local file and L_j is the value of the link. It inserts F_j into R if it doesn't already exist, setting its relevance score by $(L_j \times V_i)$. If F_j already exists in R, it's relevance score is incremented by $(L_j \times V_i)$. Thus, in cases where a file contains incoming weight from numerous snippets, its relevance score contains the sum of each individually contributed relevance score. Finally, R is sorted by relevance score and returned. Figure 4.2 depicts a visual overview.

4.3.4 Application 2: context tagging

In *context tagging*, SeeTrieve takes a given file, finds related snippets, and uses their contents to create a textual summary – or *context zeitgeist* – of that file. Unlike content

[3]SeeTrieve can use either Indri [2] or Google Desktop [1] for the content index.

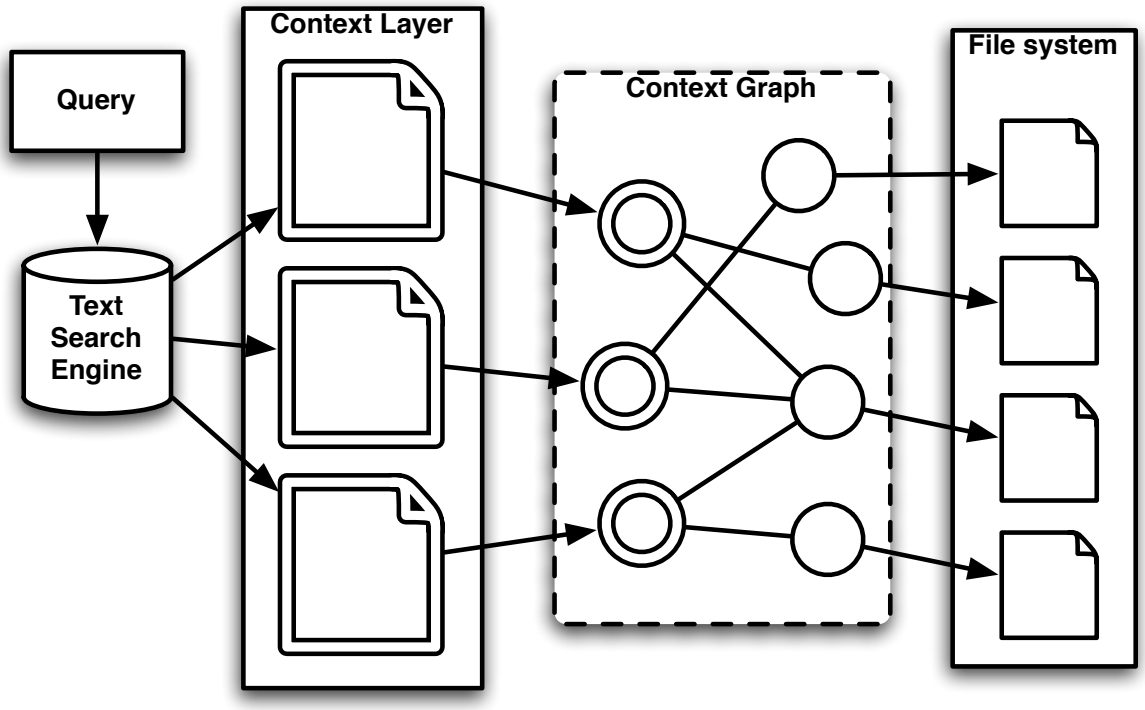


Figure 4.2: Document retrieval

classification, which uses a file's contents to identify relevant terms for that file, context tagging uses the contents of the activity that surrounds a file while it is used to identify relevant terms, offering terms that the file's contents might not even contain.

For example, an image file on a user's computer might have no useful information text content within it. Let us assume that after downloading the image from their camera, the user uploaded the image to Flickr and entered a title, description, and tags for that image through the website. Because these operations generated a set of content events surrounding the file event for the image (e.g., the time it was uploaded), they will share links with that image on the context graph. The textual contents of these events will contain useful pieces of information about the image: its title, tags, and description as entered by the user.

SeeTrieve's context tagging operates much like an inverted search. Given a file F, let P be the set of snippets related to F in the context graph. Let T be the set of tuples

$\langle t_i, f_i, c_i \rangle$ where t_i is a unique term from the contents of the snippets in P, f_i is the total number of occurrences of term t_i, and c_i the count of snippets containing t_i[4]. Let D be the set of all snippets in the context graph. Let D_t be the set of all snippets containing a term t, identified through the term index. For each $\langle t_i, f_i, c_i \rangle \in T$, SeeTrieve computes a score for each t_i using a variant of *tf-idf* defined in Equation 4.3.

$$tf_i = \frac{f_i}{\sum_{f_k \in T} f_k}, idf_i = \log \frac{|D| - |P| + 1}{|D_{t_i}| - c_i + 1} \tag{4.3}$$

The effect of Equation 4.3 is to treat the set of snippets P as a single logical snippet. Thus, it calculates term frequency (tf_i) across the contents of all snippets in P, and calculates inverse document frequency (idf_i) as if all of the snippets in P were removed from the corpus and replaced with a single snippet containing the term.

SeeTrieve calculates the final *tf-idf* score for each term as the product of the term's *tf* and *idf* values, sorts the terms by their scores, and returns the list of terms as the file's context zeitgeist.

4.4 Evaluation

The goal of our evaluation is to determine the effectiveness of SeeTrieve's two applications: task-based document retrieval and context tagging. Unfortunately, unlike traditional content-based retrieval and classification, context-based tools require that users interact with the data in realistic usage scenarios in order to gather the necessary traces, ruling out the use of an existing document corpus. Thus, our evaluation employs a two-phase user study in which users first interact with a data set while being traced by SeeTrieve and then later are asked to evaluate SeeTrieve's two applications with respect to that data.

[4] *Stop words*, or words considered too common to be useful in retrieval, are omitted from T.

All of the users in our study executed the tasks on the same machine, using the same account. Limiting the scope of the content or context information to a single user would trivialize the task of finding related data (since all available data would be relevant). We decided to merge the traces of several users simulates a single user performing a set of similar tasks over a longer period of time, allowing tasks from different users to become sequential.

First, let us step back and explain a nuance in the SeeTrieve system. The tracing component of SeeTrieve records file and UI events, periodically purging the record to log files. The processing component reads these logs and modifies the context graph from their information. In the intended form of SeeTrieve, the processing component reads these log files immediately when they are created. However, it is straightforward to decouple the tracing and processing components, allowing us to merge all trace files into a single file, then running the processing component on this merged trace file. This merged trace allowed all file activity to manifest to a common context graph, enabling links to be built between snippets and files from different users.

In our evaluation, SeeTrieve began context building immediately when a user started their first task, and ended when their last task finished. Times for the log files were adjusted such that the end time of a user's task immediately preceded the start time of the next user by a single second, pushing all other time stamps from the adjusted log backward accordingly. This allowing events from one user's task to be present within a context interval of the following user's task (and vice versa), more faithfully emulating a single user switching between tasks.

4.4.1 Study design

Phase one

Phase one is the experiment described in Section 3.4.1. The data collected during that experiment was used for this one. The experiment was originally designed for SeeTrieve although it was used for additional data in the Confluence experiments. In this study, tasks were designed to meet three criteria. First, tasks needed to be representative of typical computer file interaction, comprising different types of activity (e.g., reading, writing, information seeking, and skimming). Second, tasks needed to include files which were not all traditional, local documents. For example, a *wiki* page, which is a web page that is directly editable by users, allows the same forms of interaction historically limited to local files, but is neither saved locally nor directly modified by the user's applications (the web server moderates access). This criterion illustrates the important ability of SeeTrieveto enrich file retrieval even as emerging file interaction paradigms (e.g., web-based applications) challenge the assumptions of traditional personal information management systems. Finally, to recreate naturalness in a controlled study, users needed to have freedom to interact with files of their choosing from an uncontrolled source (e.g., the web).

The study included **15** users, **12** of which completed Task 1, **13** of which completed Task 2, and **10** of which completed both Task 1 and Task 2. The discrepancy in numbers was due to a number of factors. The tracing software failed a small number of times for one task but not the other, and some users only had the time to accomplish one of the tasks. Each task took users between 20 and 45 minutes to complete.

Phase two: Retrieval

We evaluate two aspects of SeeTrieve's document retrieval: task-based retrieval and known-item retrieval. Task-based retrieval, specific to context systems, returns items

related to the task described by the user's query. Known-item retrieval, the traditional form of document retrieval, returns a single item desired by the user. Both of these are forms of re-finding searches.

Evaluating known-item retrieval is straightforward. We directly compare the ability of each system to find the file corresponding to the user's query. Evaluating task-based retrieval, we were interested in the following: given a user's ability to issue a query that can retrieve *some* file from a particular task, how well is the system able to produce results from other files in that task?

Three to seven days after completing phase one, the volunteers were asked to return for the retrieval task. When working on the first phase, the volunteers were not informed that they would later be asked to retrieve the files. At this point, we asked them to try to remember of the files they used during the first phase.

Because most users would be unfamiliar with task-based retrieval, we felt that asking them to perform and evaluate such retrieval tasks might introduce a bias toward SeeTrieve. Instead, we asked users to locate each document used in a task by performing known-item retrieval through a modified version of *Google Desktop*, a traditional content-only desktop search tool. For the trip report task, this included the wiki page containing their report along with each paper they summarized. For the photo album task, this included each photo file, each Wikipedia page, and the page depicting their album[5]. Users were allowed to issue three queries for each document with the goal of generating a query that would return it as the first result.

The modification to Google Desktop prevented any information about the file except for the nondescript file name to be displayed in the result list. This was to prevent users from refining queries by seeing snippets of results. 30 results were displayed per page.

[5]Note that although the web documents were not stored locally, Google Desktop indexes the browser cache, allowing users to identify terms that would successfully retrieve the page as if it were local.

In these experiments, SeeTrieve was parameterized with a 30 minute context interval, allowing any text viewed within 15 minutes of a file to be related. For task-based retrieval we report task recall and task precision values. Each query Q_i was produced for a document from some task T_j, for which the search engine produced a result list R_{Q_i}, sorted in ascending rank (where lower rank indicates higher quality). For each document's query, we measure the task recall as the percentage of local documents in R_{Q_i} that are also contained in T_j, with $|T_j|$ being the number of documents in T_j:

$$task\ recall = \frac{|R_{Q_i} \cap T_j|}{|T_j|} \tag{4.4}$$

We measure task precision in terms of R-precision, a metric which reflects the portion of the first N results (i.e., the best N results) which are from T_j, where $N = |T_j|$:

$$task\ precision = \frac{|first_{|T_j|}(R_{Q_i}) \cap T_j|}{|T_j|} \tag{4.5}$$

To even the comparison between SeeTrieve and Google Desktop, we filtered the results of Google Desktop in two ways. First, we remove results that were not accessed at least once by a user (and hence, would not be within SeeTrieve's index). Second, when comparing Google Desktop and SeeTrieve, we remove any web-cache results. Because SeeTrieve only indexes local files that have been accessed and the retrieval task only considers local files as correct results, to include other files (e.g., non-accessed files or web-cache results) would unfairly penalize Google Desktop. We also exclude results from SeeTrieve for files within known system directories (e.g., *Local Settings*), as Google Desktop considers these files irrelevant, and to include them in SeeTrieve would unfairly penalize it.

We report two tables of task-based retrieval. In the first table the task-retrieval results, we removed queries for which neither system could produce a single task file.

	SeeTrieve		Google Desktop	
Task	Recall	R-precision	Recall	R-precision
$Task_1$	0.945	0.667	0.314	0.282
$Task_2$	1.000	0.730	0.177	0.163
$Task_{all}$	0.964	0.689	0.267	0.241

Table 4.1: Task-based retrieval.

	SeeTrieve		Google Desktop	
Task	Recall	R-precision	Recall	R-precision
$Task_1$	0.677	0.478	0.225	0.202
$Task_2$	0.926	0.676	0.164	0.151
$Task_{all}$	0.749	0.535	0.207	0.187

Table 4.2: Task-based retrieval, not filtered.

This is due to our task recall metric; we are interested in how well a system could return task files given the user's ability to recall any file from that task. Queries which produce no files from the task indicate that the user was unable to recall a file from the task, and hence is unlikely to benefit from task-retrieval. Table 4.1 lists the task-based recall and R-precision values for both SeeTrieve and Google Desktop, in this scenario. For task-based retrieval SeeTrieve achieves nearly 100% recall with the same precision as Google Desktop. This indicates that users could retrieve any document used in a task by remembering just one document from that task. Even in the case of remote documents (e.g., the wiki page) this holds true, highlighting SeeTrieve's ability to utilize information from any source when retrieving local data. Note that given the task sets were small, 4-7 documents, a precision of 50% indicates that all of the documents would be listed in the first 15 results.

These results reflect only scenarios in which users are able to find at least one item from the task, evidenced by the presence of at least one task file within a query's result list. We report the recall scores which include queries that do not return any task items in Table 4.2.

	SeeTrieve		Google Desktop	
Task	Recall	MRR	Recall	MRR
$Task_1$	0.701	0.388	0.657	0.555
$Task_2$	0.963	0.545	0.519	0.491
$Task_{all}$	0.777	0.433	0.617	0.536

Table 4.3: Known-item retrieval.

We believe that the results of SeeTrieve's task-based retrieval should be considered in isolation. Because Google Desktop was not designed with task-based retrieval in mind, a direct comparison against SeeTrieve is less meaningful. Furthermore, in the retrieval task users issued queries intended to recall individual items. Had they issued queries to find as many familiar items as possible, their search strategies might have been more general.

Table 4.3 lists the known-item average recall and average mean reciprocal rank for both SeeTrieve and Google Desktop. As compared to Google Desktop, SeeTrieve recalled more items but, on average, positioned those items slightly further down the result list. This illustrates two results. First, despite the increase in average position, SeeTrieve placed results well within the first page of results, providing some evidence that its known-item retrieval could compete with those of content-only retrieval.

Second, SeeTrieve found documents when Google Desktop did not, especially in the image retrieval task, again showing the relevance of user-interface text when applied to content-free data. For example, a search for "James Gleick" through Google Desktop was unable to retrieve the image file "log1.jpg" because neither the contents nor the name of the image were relevant, while the same search in SeeTrieve was able to retrieve the image. In cases of PDF recall, the slight improvement in recall was largely due to users unknowingly placing too much information in their query for Google Desktop to work. For example, a search for "hierarchy projection paper" failed in Google Desktop because the term "paper" was not present in the document itself, though present in

the context (e.g., the wiki summary was titled "paper review").

The success of SeeTrieve in task-based retrieval shows that (a) as an element of task, a document contributes some content to that context, (b) a query that identifies a document can also identify the context of which it is a part, and (c) a query that identifies a context should identify all files which were used as part of that context.

While users worked with specific applications in this experiment (e.g., a PDF reader, a browser), it is important to note that the way in which context was collected and applied was application independent. Had the users been instructed to report their summaries in an email rather than a web page, the text they generated would have still been available to SeeTrieve and useful in retrieval. Given the contents of this email would be acquired by its screen text rather than its file contents, SeeTrieve's access to the information would persist regardless of whether users' emails were through Outlook or Google Mail. Hence, SeeTrieve enables context retrieval without making any assumptions about applications beyond the fact that they must eventually present text that is meaningful to the user through the UI.

Phase two: Classification

To test SeeTrieve's context tagging, we need to show that the terms it identifies as relevant are familiar to the user of that file. We chose one local file at random from each of the two tasks (i.e., one PDF and one image) for each user and generated a zeitgeist for each file using context tagging, which we term the *context* zeitgeist. We also placed each of the PDF's into a single content index using Indri [2], and asked it for the set of keywords it considered most relevant for each PDF, creating a content zeitgeist for each PDF, which we term the *decoy* zeitgeist. In these experiments, a context interval of 10 minutes was used. We then presented users with five zeitgeists

plato, ***philosopher***, ***socrates***, **album**, thumbnail, item, **subalbum**, **upload**, file, ***hegel***, ***bc***, ***philosophy***, ***athens***, **photo**, platon, ***kant***, use, time, **caption**, **wikipedia**, size, add, sort, ***ancient***, default, ***greece***, edit, apply, description, ***oracle***, summary, administrate, megabyte, set, ***philosophic***, ***argue***, date, option, create, ***western***, charge, **gallery**

Figure 4.3: Example of a user's actual context zeitgeist. This zeitgeist was produced for an image from a user's photo album task for the topic "philosophers". ***Bold italicized*** words describe the topic, **bold** words describe the task, and the underlined word was contained within one image's file name.

for each of their two randomly chosen files[6]. For the PDF we presented the context zeitgeist, the decoy zeitgeist, and three other randomly chosen context zeitgeists for other files not accessed by that user, which we term *incorrect*. For the image, we presented the context zeitgeist, and four incorrect zeitgeists (i.e., those pertaining to files from other users' tasks). We asked the user to rate each zeitgeist on a 3-point Likert scale, where 3 indicates that the terms describe the file well, and 1 indicates that the terms are irrelevant for the file.

Figure 4.3 illustrates an example zeitgeist produced for an image chosen by a user during the photo album task on the topic "philosophers." We draw three points from this example. First, 15 of the first 20 words are relevant to the file, either describing the topic, task, or filename. Second, both the topic of choice, philosophers, and the source of information, Wikipedia, are represented in the zeitgeist, either of which the user may recall when trying to retrieve an item. Third, many of the irrelevant words are included because there is not enough overall system data to exclude them. For example, words such as thumbnail, item, add, sort, administrate, etc. could reduce in significance as a user interacted with the photo album software during other tasks, as their discriminating value might weaken.

[6]To avoid triggering memories with users for the retrieval evaluation, this phase always followed the retrieval evaluation, typically by 1-2 days.

Task	Target	$\bar{\chi}$	σ	t-test	u-test
1	Correct	2.50	0.80	—	—
1	Decoy	2.08	0.79	0.213	0.092
1	Best Incorrect	1.58	0.79	0.010	0.008
2	Correct	2.91	0.30	—	—
2	Best Incorrect	1.27	0.47	<0.001	<0.001
1 + 2	Correct	2.70	0.63	—	—
1 + 2	Best Incorrect	1.43	0.66	<0.001*	<0.001*

Table 4.4: Classification results. The mean scores of *decoy* and *best incorrect* are compared to the mean score of *correct* using the t-test and u-test. The P-values from these tests are depicted in the final two columns. Note the t-test and u-test values in the combined task reporting are heavily influenced by the significance values of task 2.

Table 4.4 lists the results of our classification experiment. When considering the incorrect zeitgeists, we took the highest scored incorrect zeitgeist for each user's task and averaged that score across users. For example, if a user for task 1 scored the incorrect zeitgeists 1, 1 and 2 respectively, we considered 2 as the best incorrect score and averaged those scores across users for task 1. For each zeitgeist, we present the average score, standard deviation, and P-value as calculated by the Student's t-test and Mann-Whitney U test between that zeitgeist and the context zeitgeist. We show the results for each task, and the average across both tasks.

We draw three conclusions from these results. First, the context results are significantly better than the best incorrect result in all cases, indicating that context tagging is successful. Second, the context results in Task 1 perform as well the decoy results[7]. This indicates that SeeTrieve's snippets are able to capture the relevant text of an indexable document at least as accurately as document content alone. Third, the context results for Task 2 are extremely accurate, achieving an average score of nearly 3. This indicates that SeeTrieve accurately classifies documents that contain no indexable terms at all, an impossible task with traditional content-based schemes.

[7]Although the average score is higher for context, we did not have enough users to show a statistically significant difference, as evidenced by the p-value.

We believe SeeTrieve classification could be applied in cases where users have documents whose origin or use they do not recall. For example, when discovering an unfamiliar document in a long-before used folder, enabling the user to see important words from the surrounding activity might reveal important insight (e.g., the paper was downloaded in a previous literature review).

4.4.2 Summary

The evaluation of personal information management tools remains a difficult problem [14, 32]. While our evaluation was designed to illuminate the abilities of our system, we believe there are lessons from the evaluation that could constitute a contribution to personal information evaluation. Hence, we detail the advantages, disadvantages, challenges, and nuances in our approach.

Controlled vs. Field study

The effects of time and data set size are important to consider in personal information management. In a controlled study, there is the danger of too little data, making retrieval tasks trivial or uninteresting. This is less likely in a live deployment; however, building a sizeable data set for a single user requires a large amount of time and may be impractical.

We address this problem by having multiple users share the same computer at different times to simulate the effect of a single user working on multiple, similar tasks. While this enhances the amount of data that can be collected within a limited time period, it introduces a challenge during retrieval that users issue queries on a corpus that contains large amounts of "personal" data of which they are unaware. This blind spot occasionally manifested in unsuccessful queries (e.g., a user searches for "ACM pdf" on a system containing over 100 ACM papers). We addressed this problem both

through the design of the data creation tasks and the design of the retrieval task.

Data creation

When designing our phase one tasks, we considered two points. First, that users should be made aware of the expanded corpus of "personal" data, outside of the files they directly interact with. Second, that because the other data on the machine is unknown to the user, the users should have discriminating information about their files that they can use to avoid overlapping terms that might, unintentionally, retrieve another user's data, as this would not occur if a single user performed all tasks.

To inform users of existing data in the paper review task, users selected their papers from a local folder that was populated with a large number of papers. This tacitly communicated that they might need to use more specific keywords when later retrieving the document. To prevent overlap in the paper review task, papers that were read by one user were removed from the papers directory after they completed their summary to avoid two users sharing the same item. In the photo album task, each user was assigned a unique topic area to research, resulting in distinct sets of relevant keywords among users.

Retrieval

We used query refinement during the retrieval task to further mitigate the problem of unknown "personal" data. For each document, users were given three opportunities to generate a query that placed the document as the first result in the list.

Although this approach improves a user's ability to isolate his or her own files, we also want to prevent users from utilizing information within a set of results to refine their query, as this context information, if used in a query, could falsely boost the results of SeeTrieve. Although use of context information independent of the query results is

our expectation, we did not want to guide users to use context information during query refinement. To ensure this we developed a thin wrapper to Google Desktop that reveals only the file name in the result, and all papers were given cryptic names from the beginning to ensure that author or title information could not be derived from the results.

4.5 Discussion

It is important to consider not only how the systems differed in performance, but *why* they differed. In this section, we identify the major cases in which the content based system was not amenable to the way in which the user recalled their document.

4.5.1 Implicit linking

During their retrieval task, many of the users reported — especially when unable to recall documents — their preferred retrieval method would have been to first find the wiki page or photo album page, which typically linked to the forgotten documents. This illustrates a case in which users would have applied relationships between documents as a retrieval tool. Since SeeTrieve was able to implicitly recreate this linking through the user's activity, it was successful in retrieving task files even when the user forgot enough details about them to form a successful content query.

4.5.2 Abstract vs. detailed recall

Of the 12 users who completed the paper reviewing task, only 2 were able to recall all three of the reviewed papers. Users were more successful with the photo album task, with 9 of 13 users recalling all three photos used. The most apparent quality observed was that in most cases users remembered their documents abstractly rather than in

detail. Below, we discuss two areas where this tendency manifested.

Author vs. Reader

Although users tended to forget at least one read paper, in almost every case they recalled the summary page they created. We attribute this to the personalization derived from authorship.

Many users recalled keywords from the paper they had placed on their summary pages, and used them as queries to retrieve their papers. This suggests that users place words in their summaries that they believe are effective descriptors of the read document, and, by identifying these words and using them in authoring, are more likely to recall them. In practice, these descriptors were very effective in retrieval.

Also, users often remembered words that were of their own origin and successfully applied them in searches for their summary page. For example, a number of users recalled words from the unique or clever title they produced for the document.

We believe that the act of authoring summaries forces users to engage with the documents they read in more depth. When summarizing, users choose words that intuitively describe the document to them; these words reflect the user's own concept of the document and do not necessarily have to exist within the document itself. Because users are more prone to recalling information through an authoring task, we believe that this activity of recollection, not accomplished well by existing content based systems, is important to support, and is supported well by SeeTrieve in this experiment.

Topic vs. Item

In the photo album task, every user recalled the broad topic for which they acquired images, and were able to retrieve their album web page in every instance. However, they were not always able to retrieve each individual photo. We believe there are two

reasons that explain this observation.

First, as in the paper review case, there were instances where an item within the topic was forgotten. Some of the users were able to remember specific aspects of their topic that lead them to originally choose to research the forgotten items. These aspects were usually captured in the summary of their photo album. For example, one user, given the topic of "dinosaurs," selected a specific time period within which to select particular instances. In this case, a search for "Triassic period" retrieved their photo page.

Second, users often remembered specific items without recalling important features of those items that would be necessary for retrieval. One user, researching politicians for the photo album task, recalled a specific politician whom he or she chose but could not recall the exact spelling of the name, preventing a successful retrieval of the image file. When trying to recall the web page from which the photo originated, the user applied alternate information about the candidate, issuing a successful query including the state which the candidate represents. This was an application of knowledge *about* the item that simply did not exist *within* the item itself.

These scenarios indicate that users are more likely to remember broad topic information about an object than specific details about it. By capturing task-based context information, SeeTrieve is more likely to contain keywords a user is likely to remember, improving document recall.

4.5.3 Limitations

One disadvantage of our approach is that our volunteers were not using actual personal data. Because the data from every user was being indexed by the same tool, we exerted control over the documents and topics as necessary to prevent too much content overlap. For example, if participants organized their own photos in the photo album task, it is

possible that they could have generated strongly similar activity (e.g. making "beach vacation" albums) which would have diminished our ability to measure context retrieval. We believe this impacted the recall of items, especially in the paper review task where users often reviewed papers outside their interest area. This limitation could have been overcome by allowing users to offer their own set of interesting but unread papers, with us rejecting any item that matched a previous user's selection.

Our approach assumes that user attention is limited to the application window. Clearly this is a simplification, as the very reason for multi-windowed environments is that users can view multiple sources of information simultaneously. SeeTrieve could be adapted to include information from all visible on-screen text, perhaps more heavily weighing the text coming from focused windows. An interesting study would be one in which user's personal window interaction is examined.

We designed tasks for users to accomplish to exert some control over the study, rather than observe users working with their own tasks and files. This biased users away from multitasking, i.e., overlapping their work. Though we believe that tasks are generally concentrated enough that temporal locality is useful (evidence is presented in chapter 3 as well as [43, 47, 53]), our experiments involve a relatively clean segmentation of disparate tasks. This is likely to have improved the results we observed, relative to a naturalistic approach.

Volunteers in this study did not have familiarity or interest in the tasks in which they engaged. This likely influenced the results we witnessed. In particular, we believe that users' rates of forgetting are stronger in these tasks than they would be in tasks in which they were familiar with information.

Another limitation is the lack of long term data, which involves two issues. First, we do not see how SeeTrieve works in supporting recall tasks on older data. For example, it would be useful to see the effect of longer periods of file disuse on users' queries for

their files, as memory decay is stronger. Second, the algorithms may need to adjust to the scale of a larger data set. For example, the *tf-idf* values for a file's snippets will be affected by the continued growth of the snippet corpus, meaning the words determined by SeeTrieve to describe that file could evolve over time. Problematically, this occurs even in the absence of new events on that file.

4.5.4 Concluding remarks

An important lesson in this work is that users are generally more able to recall the context in which a file was used than the contents of the file itself. One of the primary reasons for this is that this context often contains information about the personal ways in which a user conceptualizes a document.

In the process of doing a literature search for a research paper on contextual retrieval, one might issue the query "papers on contextual retrieval", to which a search engine like Google might be able to return papers on a conceptually similar topic like "personalized search". This retrieval is enabled in part by the fact that the hyperlinked structure of the web can leverage the multiple ways in which the universe of users organizes information. For example, an individual might link to a "personalized search" paper within their "context retrieval" web page, enabling search tools to connect the similar concepts. In local document retrieval, this structure cannot be leveraged. However, being able to connect the user's initial query to the document which was ultimately retrieved through a system like SeeTrieve allows the user to implicitly describe their own documents through their behavior.

Chapter 5

Passages: tracing text as a first class entity

The *Passages* system enhances information management by maintaining a detailed chronicle of all the text the user ever reads or edits, and making this chronicle available for rich temporal queries about the user's information workspace. Passages enables queries like, "which papers and web pages did I read when writing the 'related work' section of this paper?", and, "which of the emails in this folder have I skimmed, but not yet read in detail?" As time and interaction history are important attributes in users' recall of their personal information, effectively supporting them creates useful possibilities for information retrieval. We present methods to collect and make sense of the large volume of text with which the user interacts. We show through user evaluation the accuracy of Passages in building interaction history, and illustrate its capacity to both improve existing retrieval systems and enable novel ways to characterize document

activity across time.

5.1 Introduction

We interact with our desktops through applications' graphical user interfaces, through which large amounts of text are presented to us. This text can be captured and cheaply stored, making it amenable to indexing and retrieval. This work explores the application of this text to information management and retrieval, specifically by capturing the fine details of the user's interaction as a first class entity.

There are two important attributes of the viewed desktop text — which I will refer to as the *text stream*. First, it contains a comprehensive record of all our text-based desktop activity, including the contents of all the web pages, emails, and other files with which we have interacted. Second, detailed timing information about its contents' visibility is available, enabling a precise record of what was viewed and when it was viewed [26]. These attributes can be combined to form a rich history of the user's interaction with their information, documenting for every point in time what the user was reading or writing.

This is well suited to address a need highlighted by recent studies on information retrieval which show that the history of our interaction with information plays a fundamental and useful role in our recollection of that information [5, 15, 25]. For example, having recently read a useful fact from a research paper while writing a literature review for a grant proposal, a user may want to refer to that paper again but neither remember its location nor any specific keywords with which a search query could be issued. On the other hand, they may remember contextual, timing aspects of their interaction with the document, such as having read it within the week prior to the proposal deadline, having skimmed the document (e.g., spending under 10 minutes reading it, or only having read certain parts), or having used it contemporaneously to the grant proposal

within which they wrote about the lost document.

This recollection of temporal events is very nuanced and personal; yet existing systems and applications such as browsers, email clients, and filesystems, remain coarse and one-dimensional in supporting it. Although some research systems have addressed this limitation by supporting time from the ground up, they lack generalized applicability as they involve either a dramatic overhaul of existing systems (e.g., [17, 28]), or application-specific adaptations [49]. Our work captures the best of both approaches, being an application-agnostic adaptation of existing applications and filesystems to support rich time-awareness; in essence, migrating the state of the art from proof of concept to usable implementation. Further, our approach does not adhere to a strict definition of a file: where existing systems treat information by distinct file types (e.g., web pages, emails, or PDFs), Passages's tracing at the text level captures information interaction without rigid, foreknown types. Hence, our approach is useful in new interaction contexts such as web-based application interaction, where traditional document definitions do not apply.

We designed the Passages system to capture the user's text stream and transform it into a rich, finely grained, application-agnostic, information-interaction history for use in information retrieval solutions. Passages can answer questions that are not easily answered by existing systems, such as, "which of these conference papers have I not yet thoroughly read", "which documents did I read when writing this literature review section", "what functions was I working on before I committed this code", and "which documents did I spend the most time on the month before the grant deadline?" In this chapter, we detail and address the challenges involved in transforming the raw text stream into a form from which useful temporal information can be drawn. We show that these methods are accurate, efficient, and substantially improve upon existing systems.

5.2 Passages

The Passages system has two main objectives: chronicling the user's interaction with their document workspace in a granular fashion, and making this interaction history available in retrieval scenarios for which existing systems cannot produce effective solutions. Because of the complexity and highly personal nature of time, the aim of Passages is to provide a set of primitives through which complex, personal queries can be constructed, rather than to define *a priori* temporal queries for which we tailor solutions. Hence, the system is designed to be a platform from which time-aware management systems can draw information to satisfy novel information needs. For example, a desktop search tool could programmatically interface with Passages to discover files which were more active and prune for files which have never been used. This chapter focuses on presenting and evaluating the temporal tracing and retrieval infrastructure.

Our description of Passages will continue as follows: first, we outline some example queries which Passages was designed to handle, but which existing systems cannot easily implement; we follow this with an overview of the design of the Passages system; we then revisit these queries with their respective implementations within Passages; finally, we describe why existing systems are unable to adequately implement these queries.

5.2.1 Queries about activity

The following four information needs are not an exhaustive listing of the functionality of Passages but rather intended to serve as examples of the rich spectrum of temporal queries Passages was designed to support.

1. *Finding the "to-read" stack.* Tasks often need to be deferred until a later point. Sometimes, we come across research papers which we find initially interesting but to which we are unable to dedicate immediate attention. Other times, we receive a long

email which we skim but do not read in detail and to which we need to eventually respond. After the initial filing, we may wish to revisit these files. A useful query, then, would be one which identifies files in our workspace which we have skimmed but not completed.

2. *Task-based retrieval.* Document interaction takes place within a task, and tasks usually include other files or sources of information. Often, a user remembers the surrounding task of a misplaced document, and may remember other files used as part of that task. For example, the process of writing a literature review involves collecting, reviewing, and summarizing other papers within the field. A research paper may be related to many different files; the experimental section was authored alongside some of the source code files developed for the project, while the literature review section is related to the different research papers read while writing the review. The query for this example would be to find all files used while the user worked on the related work section.

3. *Pattern-based activity.* Users' natural recollection of a file's activity is more complex than filesystem attributes such as "last-accessed" [5]. For example, a user may forget the location of a paper they authored, but remember the time period before the conference in which they worked on it, that they worked on it for long periods of time, and that they frequently worked on it. This recollection involves multiple applications of time, from a range of when interaction occurred, the frequency of interactions, and the typical duration over which this interaction occurred.

4. *Information Provenance.* Often, a user's document is a composite of various sources of information. For example, a document of notes for a literature review may contain passages or paragraphs from different papers which were part of the review. A source code file may contain functions or other portions copied from other files. After a document becomes mature, a user may wish to recall the sources from which a part of a

document (e.g., a paragraph) was derived. Provenance-aware storage systems motivate and attempt to address this problem (e.g., [41]).

5.2.2 System design

Passages comprises two subsystems: *tracing* and *retrieval*.

Tracing

The tracing system records event streams from two system sources: the graphical user interface (Gui) and the filesystem. Tracing user behavior at the Gui layer involves recording *focus events*, which occur when an application window gains focus. This typically occurs when the user activates a new window by clicking in its visible region, or by minimizing a previously active window.

When focus events occur, the visible text contents of the window — which we will refer to as a *snippet* — are acquired and added to a persistent queue. This snippet is tagged with information about the focus event, such as the time at which the event occurred and the identifier of the window's application. If some small duration transpires with no new focus event, the tracing system acquires a new snippet from the active window; this handles cases where a given application window gains new content (e.g., a browser surfing to a new web page).

Filesystem tracing involves recording operating system file calls, such as *read* and *write*, including the file names on which these calls occur, the calling application, and the event timestamp. This occurs through existing operating system tracing facilities which expose access to third party code, such as *detours* for Windows[1] and *dtrace* for Mac OS X [2].

[1] `http://research.microsoft.com/sn/detours/`
[2] `http://www.sun.com/bigadmin/content/dtrace/`

One of the primary goals of Passages is to operate with minimal system assumptions and avoid application-specific design, which would require retrofitting countless applications to maintain granular records of the information they display. As such, the tracing system is purposefully low-level. In our approach, text is acquired through the system accessibility API, which allows a window's graphical components (e.g., text areas, buttons) to be traversed by a third party application. This information is shallow; it does not allow, for example, the third party to determine that a given text area is displaying a particular file. Furthermore, there is no context available; for example, if a window displaying text is scrolled to different positions at times T_i and T_{i+1}, there is no way to determine that the visible text at these two points are merely different views of the same source data. Consequently, the text stream is undifferentiated, redundant, and difficult to reason about; this is one of the essential challenges Passages addresses.

Conceptually, making sense of the text stream is like a computer vision problem; given a raw input stream, the task is to identify persistent objects (e.g., paragraphs or document portions) within their surroundings (e.g., peripheral text, such as advertisements on a web page) in different orientations (e.g., a snippet may only display a portion of the text of interest). Our approach is to break the snippets into small, atomic, uniquely identifiable text units. We accomplish this through *landmark chunking*.

Landmark chunking

Landmark chunking is a process which separates a sequence of text into smaller subsequences, or chunks [42]. Using parameter D, a fixed width sliding window of width W is run across the file contents, character by character. At every character position k, an efficient fingerprint algorithm[3] is applied to the contents of the window to achieve a value F_k. If $F_k \ mod \ D = 0$, position k represents a chunk boundary (see Figure 5.1).

[3]The Rabin's fingerprint algorithm is used for this purpose [46].

Our work uses the TTTD algorithm, which is tunable by average chunk size C and window size W (D, which represents the frequency of chunk boundaries, is set accordingly to reflect these values) [16].

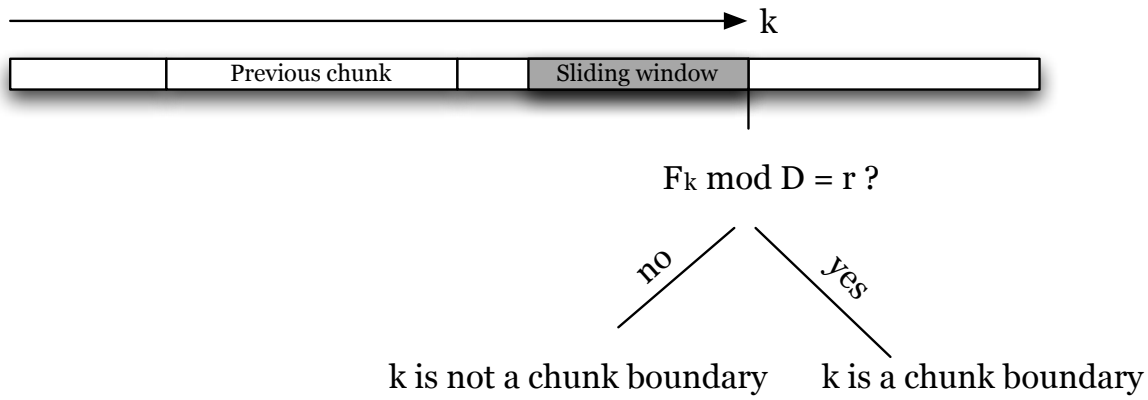


Figure 5.1: Sliding window

The critical property of landmark chunking is that the boundaries it creates are based on the local contents of text rather than a fixed width. In a fixed width approach, inserting a single byte into the contents will affect the chunk boundaries of all subsequent text. With landmark chunking, the chunk boundaries identified for a given subsequence of text are resistant to changes in that subsequence's surrounding text; with high probability, an edit to a set of text will only affect the chunk boundary which follows the edit (although often it is the case that edits do not occur within the fingerprint window, and hence do not affect the boundaries at all). We detail the importance of this property later.

At the point in which a snippet is created, we break its contents into chunks. Each of these chunks' contents are individually hashed with the MD5 function to attribute a unique, small (128 bits) identifier to them. Conceptually, this transforms a snippet's representation from a continuous sequence of text to a set of hash values; this is called *document fingerprinting*, and is useful in tasks such as efficiently approximating doc-

ument content similarity (e.g., [18]). The chunks derived from a snippet inherit the timing information from the snippet, creating for each chunk a tuple (H, S, E), where H is the hash value of the chunk, and S and E are the points at which the chunk gained and lost visibility, respectively.

Indexing

These tuples are placed within a visibility table V saved within a relational database maintained by Passages.

The filesystem and file event stream are treated as follows. Initially, all of the user's files are chunked and hashed[4]. When events that modify the file's contents are observed (e.g., *write*), or when a new file is created, the file is marked as "dirty" and queued to be reprocessed. This process generates (H, F, S, E) tuples, where H is the hash value of the chunk, F is the path name of the containing file, S is the point at which the chunk first appeared within the file, and E is the time at which (if ever) the chunk is no longer present within the file. These tuples are placed within file table F within the database.

Retrieval

Passages supports two main retrieval modes: artifact and temporal. These retrieval modes are not meant necessarily to directly solve information needs. Rather, they are basic modes over which more complex retrieval systems can be built. Artifact retrieval answers requests for the temporal history of a unit of information (e.g., a document). An example query would be, "when was the last time I read this document, and for how long did I read it?" Temporal retrieval takes a date range and returns a listing of information units which experienced some activity within this period. An example

[4]Some application-specific file reading plug-ins (e.g., PDF) are used to maximize coverage

temporal query would be, "what files did I work on most heavily the month before the grant deadline?" Complex queries can be formed by combining these modes, as we detail later.

Given tables V and F described in the previous section, for a given chunk, it is possible to identify times in which it was active in the user interface (e.g., where the user read it) through the V table, as well as when and where it appears in the file system (e.g., the file(s) containing it) through the F table. By combining the information within these tables, we can uncover the provenance, lineage, and activity of a chunk of text as it exists within both layers. This becomes useful when we synthesize information from multiple related chunks. In the next two sections, we detail the implementation of temporal and artifact retrieval.

Temporal Retrieval. The temporal retrieval algorithm is as follows. Given a time range (T_i, T_j), such that $T_i < T_j$, the objective is to determine which files were used within it.

1. Chunks experiencing activity within the range (T_i, T_j) are retrieved from V, in the form of (X, S, E) tuples, where X is a chunk, and S and E are the times at which the chunk gained and lost visibility. These tuples are placed into a queue $Q_{(X,S,E)}$.

2. The tuples in $Q_{(X,S,E)}$ are converted into an activity timeline, represented as an ordered sequence of (T, X) tuples, where T is a time and X is the set of chunks which were visible at that time. Each tuple corresponds to a point in time in which the visible chunks changed (e.g., a new window gains focus, or the active window is scrolled to a different point); duration of visibility is implicit.

3. For each tuple (T_i, X_i) in this sequence:

   (a) Each chunk within X_i is looked up in F, generating a set of files containing

that chunk. These chunks are filtered to remove those which appeared after or disappeared before the range (T_i, T_{i+1}), to ensure the file matched the visible chunks at this point in time, as a file can change over time.

(b) File f_{guess} is the file which contains the highest number N of chunks in X_i. If $\frac{N}{|X_i|} \geq \alpha$, where α is a small value between 0 and 1 (serving as a similarity threshold), f_{guess} is determined to be the active file at that time period. Otherwise, no file is determined to be active and step 3(c) is skipped.

(c) The time span (T_i, T_{i+1}) and f_{guess} are added as a tuple $(F_{guess}, , (T_i, T_{i+1}))$ to a queue $Q_{(F,T)}$.

Upon completion, $Q_{(F,T)}$ contains each file which experienced activity over (T_i, T_j) as well as the time periods over which each of the files was active.

For step 3(b), if there is a tie between files (e.g., the files with the most chunks have a very close number of chunks), there is a tie-breaking routine available which involves comparing more granular chunks. Passages maintains two additional tables V' and F' which are constructed by chunking the files and snippets with a smaller window size for the landmark chunking algorithm. This results in more chunks per file, which enables the file and visible chunks at time T_i to be compared again with finer precision. Since fewer chunks means fewer rows within V and F, those tables are preferred for first round comparisons, as ties are likely to be rare. If there is no overlap between chunks of the tied files, both files are reported to be active (e.g., a split-pane widget is displaying multiple files simultaneously).

Choice of C and W depend on a number of factors. As C controls average chunk size, it determines the number of chunks produced for a given sequence of text, which affects both resolution in document comparison and storage overhead. W affects how sensitive comparisons are to minor differences, as smaller window sizes are less likely to overlap modified regions. Our work uses values $C = 100$ and $W = 20$ for V and F,

and $C = 20$ and $W = 5$ for V' and F', as snippets tend to be small (Eshghi and Tang explore the parameter space [16]). We used an α value of 0.15.

Artifact Retrieval. Artifact retrieval operates similarly to temporal retrieval, although Step 1 works as follows:

1. Given file f_i, look up from F all chunks which have at some point existed within it, producing chunk set C_{f_i}. For each chunk within C_{f_i}, its visibility history is looked up within V, producing a sequence of tuples (C, S, E) placed within $Q_{(C,S,E)}$.

The remaining steps are executed as normal, producing $Q_{(F,T)}$, from which f_i's visibility spans can be drawn.

Due to the granular nature of chunks, artifact retrieval can support more flexible types than a file: any sequence of text which produces a nontrivial number of chunks is amenable to artifact retrieval[5]. In step 3 of the algorithm sketch, at each point T_i in which chunks C_i are visible, C_i actually correspond to a specific point in a file (e.g., the one scrolled into view). So, Passages naturally answers queries pertaining to smaller pieces of a file, such as, "on which section of this paper have I spent the most time?" or "what papers was I reading when writing the related work section of this paper?"

Importantly, chunk activity information is independent of whether these chunks actually manifest in the user's filesystem. This is especially important as emerging forms of document activity — such as those enabled through web based productivity tools — bypass local storage and retain all documents on "the cloud". For example, Passages traces and maintains activity on web-based email or documents even though they never exist on the user's filesystem; since all UI information is distilled to chunks, the source does not matter.

[5]This number is dependent on C and W, but in our experiments, sequences as small as 85 created no decline in accuracy.

Advantages of Landmark Chunking

Although chunks contain content, their function is not to directly satisfy content queries but rather to be granular objects which can bear activity within the UI and can exist in files within the filesystem. Here we justify their use by relating the theoretical properties of landmark chunking to the system objectives.

Efficient comparison. Breaking files into hashed chunks enables efficient similarity comparison among documents. File similarity is effectively approximated by examining the intersection between chunk sets [18]. Subset detection is straightforward: given files f_i and f_j, producing chunk-sets C_i and C_j, respectively, when C_i is a subset of C_j, the intersection of C_i and C_j will be high relative to the size of C_i. This is important considering we are often comparing a small portion of a document (i.e., the portion which is visible through the UI) to the document itself. For comparison, a *diff* based approach would require a comparison to be executed across each snippet and each file for each query, and would be prohibitively expensive.

Reliable detection. The manner in which applications present text is highly unpredictable, causing what we refer to as the *orientation problem.* Users can scroll, zoom, and change window dimensions, but since snippets are acquired from the visible text, these variations will affect their contents (Figure 5.3). Web pages may embed advertisements within a sequence of text. Different applications may display the same file differently. For example, a local email application displays an email in addition to displaying text corresponding to a folder-organization bar and a list displaying other email subjects; while a web-based mail site may display peripheral text such as advertisements, banners, and menus (Figure 5.4).

To function correctly, Passages needs to identify and record whenever a subsequence of text S_i is visible to the user. The orientation problem arises when the variations on information presentation cause S_i to appear at different places (and sometimes, partially

cut-off) within a snippet. Since these different presentations generate different snippets with different (though overlapping) text, the chunks produced will also differ. For Passages to correctly identify the visibility of S_i requires that the chunks produced by S_i will be very similar regardless of the contents of the surrounding snippet. After all, if the chunks produced vary according to how S_i is displayed, the activity history collected will be dependent on orientation and therefore not comprehensive. Fortunately, landmark chunking uniquely affords this stability, as changes in the text preceding — and, to some extend, within — S_i will have minimal impact on the chunk boundaries identified within S_i. (Figure 5.2 shows an example.)

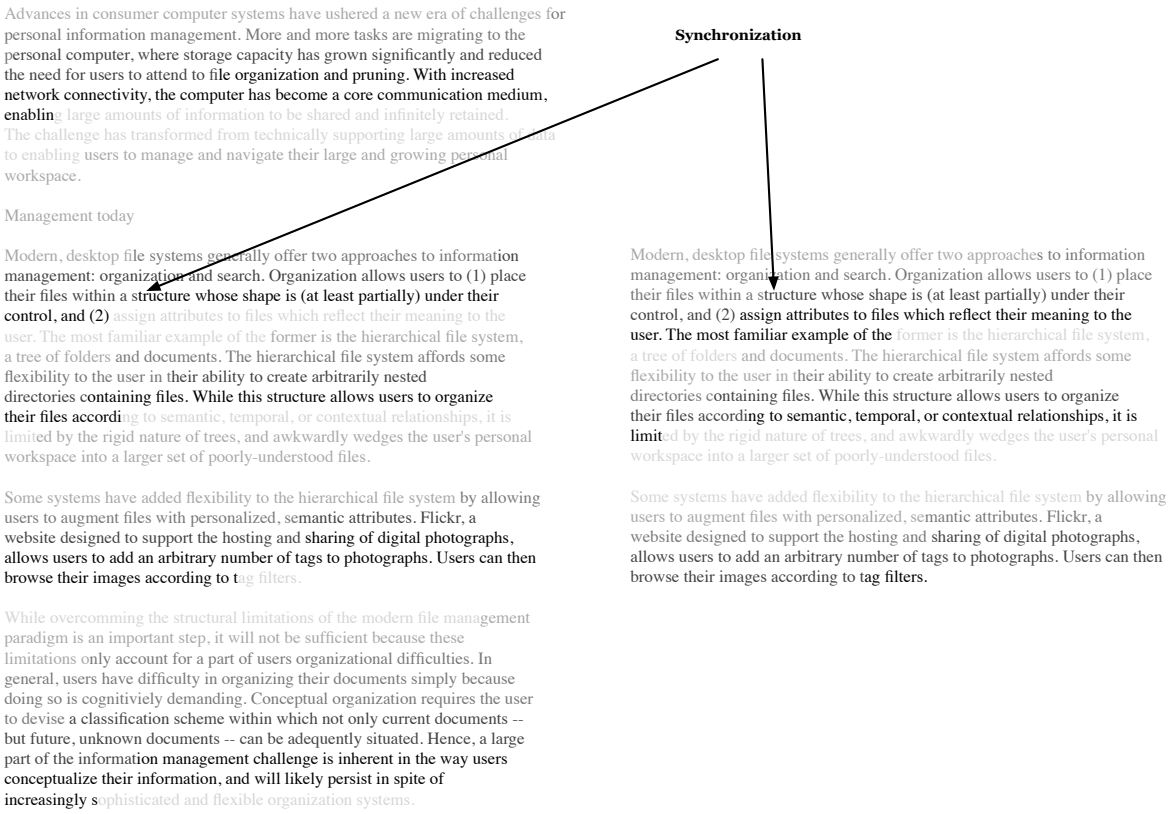


Figure 5.2: The image depicts two sets of text, with the text on the right representing the two middle paragraphs from the left. When the TTTD algorithm is run on each, the chunk boundaries are almost identical (chunk boundaries are denoted by a change in font color). In this example, 13 of the 14 chunks in the right text also exist in the left text.

Stability within filesystem. Landmark chunking is also essential with respect to the filesystem. Since inserting new data to a file will, with high probability, only affect the chunks which immediately surround the edit, many of a file's chunks will remain unchanged throughout its lifetime.

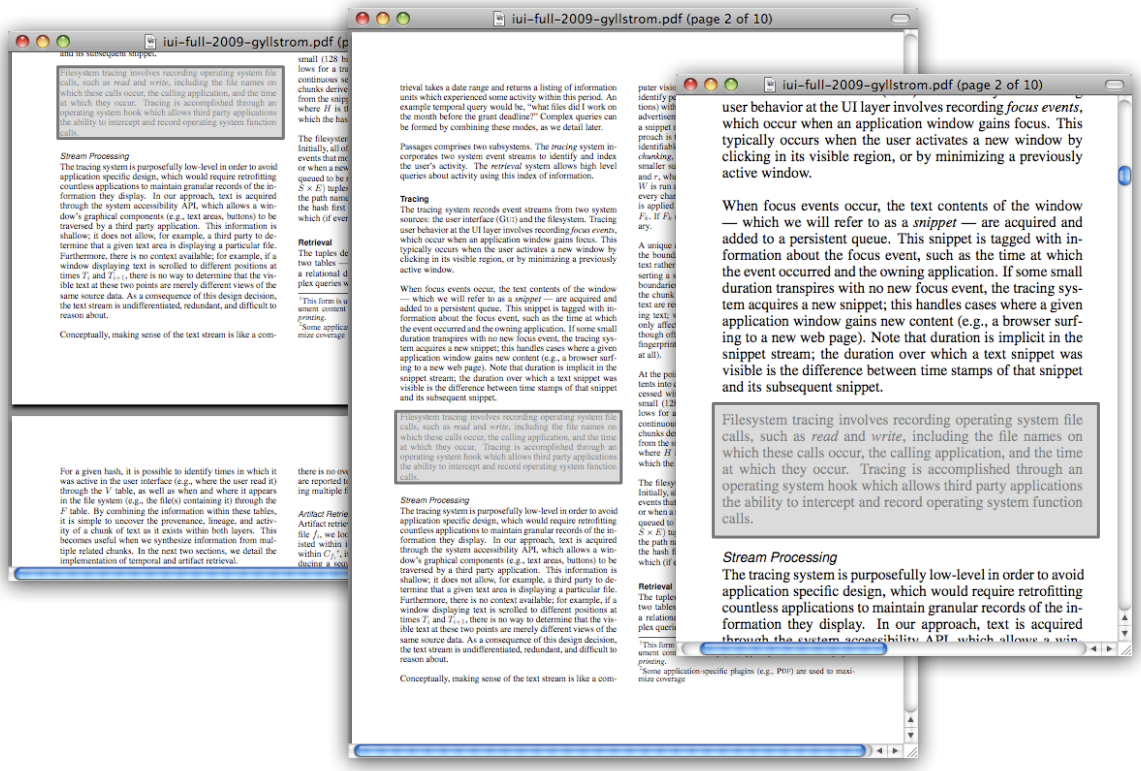
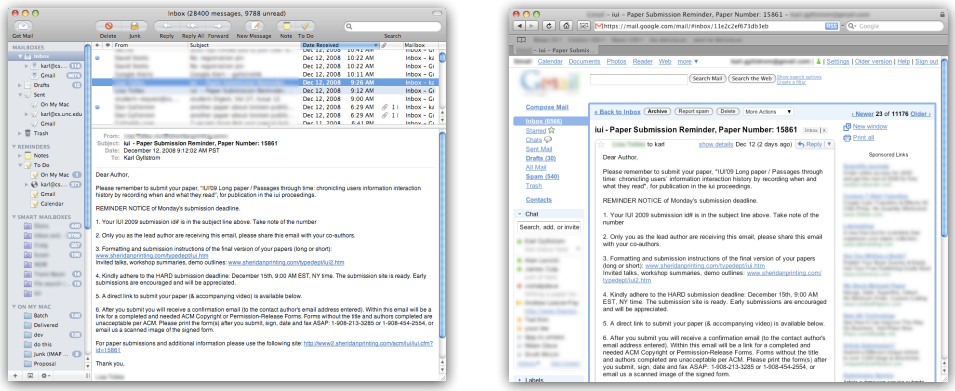


Figure 5.3: *The orientation problem.* A sequence of text (highlighted in gray) will appear at different locations on the window depending on presentation effects, such as scrolling, zooming, and window dimensions. Although they all contain the sequence of interest, the snippets acquired from the visible text will vary according to these effects, causing the sequence's surrounding text to vary.

5.2.3 Query implementations

In this section, we revisit the example information needs outlined in the beginning of this section and describe possible implementations for these queries within Passages. Although we do not outline a query language through which third parties can apply



(a) Mail application (b) Web-based mail

Figure 5.4: *Divergent displays.* Different applications display the same email differently. The mail application in (a) features surrounding text pertaining to folders and email lists. The web-based mail in (b) has advertisements, a navigation bar, and a instant message window embedded in the page.

Passages's timing information, we demonstrate cases in which the primitives available through tables V and F provide a sufficient foundation for high level temporal queries. Currently our queries are implemented through SQL queries combining V and F; though abstractions could be built upon these primitives, we instead focus on the use of chunks to highlight the flexibility from such an approach.

1. *Finding the "to-read" stack.* For demonstration purposes we will define "skim" as follows: the entire document was visible for less than $T_{document}$ seconds, and at least half of the pages were visible for less than T_{page} seconds. Resolving this query is simple: we first execute artifact retrieval on the document to determine its visibility over time; we then execute artifact retrieval on each page to determine their individual visibility durations. These values are then compared to $T_{document}$ and T_{page}.

2. *Task-based retrieval.* First, we start with the related work section, which Passages accepts as input[6] and executes artifact retrieval upon, generating visibility queue Q_v, representing each time period over which the section was visible. For each time period

[6]This would depend on the implementation of the system using Passages, which perhaps allows users to select text regions through an interface as input for temporal queries.

$T_{(i,i+1)}$ in Q_v, we expand it in both directions by some value E (e.g., 5 minutes), merging time periods in which overlap occurs; this generates expanded queue Q'_v. A temporal query is executed upon each time period $T_{(i,i+1)}$ in Q'_v, producing a set of files used during these periods. These files represent those that were used alongside the related work section.

3. *Pattern-based activity.* This query would be implemented as follows: With conference date T_{conf}, a temporal query on range $(T_{conf-60_{days}}, T_{conf})$ would return $Q_{(F,T)}$. The tuples within the queue could then be sorted by a weighting function accounting both for total visibility time and number of unique days on which it was visible.

4. *Information Provenance.* For file F_i and paragraph P_i, we would issue an artifact search on the paragraph, generating a set of files and the spans over which the chunks existed within them. We then limit this set to those files which contained those chunks previously to them appearing in F_i.

5.2.4 Problems in existing approaches

Existing systems are not equipped to enable these sorts of queries for generally the same reason: they fail to consider what the user looks at. Many modern systems and applications, including the file system, email, and browsers, employ time in some form to enhance the user's experience. For example, browsers typically enable users to view the history of pages they have visited, and some document retrieval tools record file accesses on user's filesystems. Support in these systems is generally limited by what we refer to as a "file-based" approach. By file-based, we mean that activity records are limited to when a file is created, written to, or accessed. For example, filesystems maintain time stamps for when a file was last written, and browsers maintain a record of when a web page was opened in the browser. Some systems record all filesystem events in real-time. Though common, the file-based approach is flawed for two main

reasons.

First, it lacks *temporal precision*. In practice, considering only files' accesses coarsely approximates their actual usage; knowing *when* a document is accessed is not informative of *how* the user interacted with it. The time at which an email was received does not necessarily indicate when it was read. Glance at your browsing history for a given day, and you will probably see many entries, each of which takes a single, equal position within a list. While some of these pages may have been briefly glanced at, others may have been closely examined; these nuances are lost in the linear history representation. Unfortunately, this is not simple to solve, as activity cannot be inferred from differences between time stamps; as users navigate backwards, open new tabs, switch between browser windows, and switch among different applications altogether, they significantly complicate the ability to reason about activity from access information alone.

An analogous problem occurs within the filesystem; although a file such as a PDF may generate traceable file events when opened by the user, the application may read the file to memory and no longer access the file's contents from the file system, causing the user's continued interaction with the file to be untraceable. To make matters worse, many access events have no relation to the user's activity (e.g., virus scanners operating in the background), which introduces large amounts of noise. This problem is fundamentally a consequence of the decoupling between applications — through which user-document interaction occurs — and the filesystem — where this interaction ultimately manifests.

The second flaw in the file-based approach is that it lacks *content precision*. At a given moment, the user is often more interested in a specific part of a document rather than the whole of the document. For example, the authoring of a research paper involves many different efforts, such as the execution and write up of experiments,

and the literature review. While ultimately manifesting into a single document, the document is a composite of different efforts whose conceptual overlap may be minimal, and have unique places within the user's conceptualization of them with respect to time and task. Similarly, source code files comprise functions and objects which, while related, have distinct roles which are relevant to the user at different points in time and for different objectives. Recording time at the document level prevents reasoning about individual parts of the file which may have distinct interaction histories.

Further, emerging forms of information presentation and interaction are challenging the traditional definition of documents, which severely limits the usefulness of file-based history. For example, a news aggregator web site such as *slashdot.org* displays a continually updating sequence of news abstracts from various web pages. Where the browser stores the *slashdot* front page as a single page, the front page is conceptually a collection of many different web pages; the browser's definition of the document and the user's do not agree. Hence, the history of page visits does not reflect how often or for how long one of the abstracts may have been viewed (see Figure 5.5). Similarly, a web-based mail site will have the appearance of being a single page despite enabling the user to open and read countless emails within it. Unfortunately, the ability of systems to retrieve and record interaction with information is highly dependent on how those systems define information units; the definition of the file as the atomic unit of information is too broad and inflexible.

Considering what the user actually looks at easily solves these problems. A record of what the user actually views is both more accurate and more precise than the contents of a filesystem access log or browser history [7]. To demonstrate, we show why implementing the example queries using filesystem data would be difficult or impossible.

[7] As discussed in Section 4.5.3, our assumption is that the focused window reflects the user's focus. In practice, users can view any visible portion of the screen, and often do. Our methods could be extended to include all visible text, but for simplicitly use the focused window only. We believe that in many cases, this is a reasonable approximation.

1. *Finding the "to-read" stack.* Filesystems limit file metadata to the date of last edit or access (due to application decoupling as described previously). There is no way to know for how long a file was read, nor what portions were read.

2. *Task-based retrieval.* While systems have been designed to support task-based retrieval [26, 27, 45, 47, 53], they face three limitations: (1) the limited information from the filesystem make timing information imprecise; (2) unrelated events in the filesystem cause spurious relationships to be identified; and (3) the lack of content precision makes sub-portion relationships impossible to determine.

3. *Pattern-based activity.* As with finding the "to-read" stack, nuanced activity is unavailable. Data pertaining to files read within the 2 months before the conference date may be too broad (e.g., many unrelated files were read during this period), too uninformative (e.g., some files were read extensively while others only briefly) or spurious (e.g., the file may have been read by a virus checker or search indexer). Further, the durations over which files were interacted with cannot be determined from filesystem data.

4. *Information Provenance.* Provenance would be difficult to determine, since there is no straightforward way to determine when individual portions of a file appear within it.

5.3 Evaluation

The ability of Passages to effectively answer rich temporal queries is entirely dependent upon accurately recording document activity. We evaluated Passages for the following hypothesis: given an arbitrary text sequence being displayed within the UI, Passages will, with high probability, detect it being displayed. While the document fingerprinting approach taken by Passages is formally well-grounded as a method for accurate and efficient document similarity detection [18], we wanted to ensure that the dataset which

we used was composed of real user activity, allowing for the large variety of ways in which user behavior can alter the way in which text is presented through the UI.

5.3.1 Study design

We executed a controlled user evaluation in which users accomplished a set of computer-mediated tasks involving different forms of file interaction while Passages traced their activity within the UI and filesystem. The data from this study was generated by the study described in Section 3.4.1, by executing Passages on the same raw data.

Oracle and Verification

Evaluating Passages required a reliable, *oracle* account of the user's true activity, which we built manually. Data for the oracle was collected by recording users' application window contents simultaneously to the Passages tracing system. As the Passages file tracing captured all file access events, and the evaluation machine's web browser was configured to maintain permanent web history, there was a record for the files (including web pages) that each user accessed. A local copy of each web page's in the web history were indexed by Passages; local files (e.g., the PDFs) were also indexed. The contents of windows revealed through the screen recording of a user, then, could be manually compared to the documents known to be accessed by that user within their session.

Verification occurred by directly comparing the sequence of events identified by Passages via artifact retrieval to the oracle's sequence. Recall that Passages builds an activity representation $Q_{(F,T)}$, a list of tuples (F, T), where F represents the file which Passages identifies as being viewed by the user, and T represents the time intervals over which the file was visible. We call this representation $Q_{Passages}$. For each focus event within Passages's UI-tracing record, we accessed the screen recording at the time of the event and manually identified the file contained within the focused window (when

available) to create the oracle sequence Q_{ideal}.

Additionally, we compared Passages to a file-based baseline $Q_{filesystem}$, using the file event stream generated by the users' activity; this baseline represents the approach taken by existing tracing systems and serves as an important comparison. We augmented this stream with entries from the web browser's history, which contain the points in time at which a particular web page was accessed. One challenge in this approach is that the window content and file event streams are not directly comparable, since the file and browser history trace data contain points in time in which an event occurs, and not the duration over which it occurs. For example, we know that file F_i was read by an application at time T_j, but cannot infer for how long it was read from the trace. To deal with this, for each access event F_i corresponding to file F, we created a tuple $(F, (time_{F_i} - N, time_{F_i} + N))$ and placed it within $Q_{filesystem}$ (different values of N were tested). Since this approach means many of these intervals overlap, we allowed $Q_{filesystem}$ to have multiple files considered active during a given time period.

Users and Tasks

As stated previously, we used data generated from the users in the SeeTrieve study, using the tasks described in Chapter 3. We recruited 15 volunteers from a pool including employees at the HP Labs campus and students from the Stanford University campus. Of these volunteers, 12 were researchers, 2 were staff, 1 graduate student from Stanford university, and one alumni from Stanford University. All of the volunteers completed task 2, and 13 completed task 1 as well. In total, about 14.27 hours of user activity was traced and indexed by Passages.

Artifact selection

We selected two information units to measure Passages's performance: files and paragraphs. We selected 136 files, including web pages and local files from the users' traces. We selected paragraphs from many of these files. This was done through a script which selected a random character from the file's text, after which we selected that character's enclosing paragraph. In some cases, the character did not originate from a meaningful paragraph (e.g., a row from a table); in these cases, we tried to identify a meaningful enclosing object (e.g., the table). We selected 91 paragraphs; these paragraphs ranged from a minimum of 13 words to a maximum of 290 words, with a mean of 97.8 words.

5.3.2 Results

In this experiment, we identified the systems' performance in two related areas: first, the amount of users' activity that is correctly identified by Passages — or *recall* — and second, the percentage of the file activity identified by Passages that is correct — or *precision*. In our reporting of results: T_{ideal} represents, in seconds, the total duration over which files have been manually identified to be active; T_{system} represents the total activity duration over which files have been identified as being active by a given system (either Passages or pure file tracing); and $T_{overlap}$ represents the total duration over which the system's and the oracle's activity records overlap, indicating times in which the system is performing correctly. Recall is calculated by dividing $T_{overlap}$ by T_{ideal}, capturing the amount of actual file activity that is accurately captured by Passages:

$$Recall = \frac{|T_{system} \cap T_{ideal}|}{|T_{ideal}|} = \frac{|T_{overlap}|}{|T_{ideal}|} \tag{5.1}$$

Precision is calculated by dividing $T_{overlap}$ by T_{system}, representing the amount of activity captured by Passages which is correct:

$$Precision = \frac{|T_{system} \cap T_{ideal}|}{|T_{system}|} = \frac{|T_{overlap}|}{|T_{system}|} \tag{5.2}$$

Accuracy

Method	T_{ideal}	$T_{overlap}$	T_{system}	Recall	Precision
File	11338	11224	11325	0.990	0.991
Paragraph	3958	3925	4159	0.992	0.944

Table 5.1: Performance of Passages in paragraph and file tracing tasks.

Table 5.1 reports the numbers from the Passages comparison. The system performs quite well in terms of both recall and precision, with few reported misses. We inspected failing cases in both systems: In document tracing, there were cases in which separate contained common content; for example, by copying a paragraph from a paper to the summary page (typically, when overlap occurred, common contents' surrounding text on each document was enough to differentiate the two). In paragraph tracing, there were some cases that damaged performance. One of the randomly identified paragraphs spanned the classification and term section from an ACM paper, which had a lot of content overlap with other ACM papers. Hence, it was often mistakenly detected to be active (filesystem tracing data can mitigate this problem, as cases of ambiguity can potentially be clarified by checking which of the matching files was actually accessed).

The performance of the pure file stream, as reported in Table 5.2, was substantially worse. Since file spans could overlap, we calculated recall and precision on a per file basis and aggregated results. We experimented with different values of N, ranging from 60 to 540 seconds. Higher values of N resulted in better recall, although the precision dramatically decreased. In this experiment, we removed from the stream all activity which occurred on files that were not in the evaluation set (e.g., system and configuration files), meaning a large number of irrelevant file events were pruned from consideration. Had we included them, the numbers reported for the file-based tracing

N	$T_{overlap}$	T_{system}	Recall	Precision
60	2530	17197	0.238	0.147
180	5629	48605	0.529	0.116
540	9507	131882	0.894	0.072

Table 5.2: Performance of pure file and browser history tracing. The oracle's account totaled 10633 seconds.

would be substantially worse, as a previous study has shown the majority (about 96%) of file events within the file-stream pertain to files with which the user is not actually interacting [27].

Speed and size

Passages was designed to (a) operate at interactive rates, enabling it to be incorporated into search tasks like the examples described earlier; and (b) demand few system resources such that it could be transparently integrated into the user's desktop experience. Our tracing system was built upon mechanisms designed to operate interactively. UI tracing involves accessibility support, provided by the operating system in order to enable third party assistive applications (e.g., screen readers for the visually impaired) to operate transparently in the background. File tracing operates at the kernel level. Landmark chunking, used in applications such as distributed file systems, is also designed to be efficient enough to integrate into interactive systems [42].

Since the size of our user data is relatively small, we tested the performance of the queries on much a much larger data set. To construct this dataset, we grew the visibility data table by replicating database rows by about 140 times. We chose this factor by dividing the number of work hours in a year (50 weeks times 40 hours per week) by the duration of our data ($\sim$ 14 hours), in essence expanding the database to approximate the size the it would be if a user worked diligently at their computer for a year. The visibility table grew to around 600 megabytes, which has a similar size to commercial desktop search tools like Google Desktop. Since the temporal and artifact

queries involve simple range queries and table joins, this size expansion did not have a large impact on query performance. For each file in our experiments, we executed a triple joined query — structurally similar to the one used to implement Passages's artifact retrieval — on the expanded data set using a PowerMac G5 machine from 2003, achieving an average of 3.33 seconds per query. Althought we did not explicitly keep the database in memory by keeping the database program open, we did not purge file blocks to prevent the database from being cached in the filesystem.

Consider a sample joined query where we first select all chunks which file f_i has ever contained, then search all the times over which those chunks have been visible. Assume the database is indexed on the file column in the F table, and the chunk column in the V table. The look up for the chunks contained by f_i should be reasonably fast: f_i is a fast lookup, and the chunks will be easy to find because the column is sorted by f_i, hence the chunks will be contiguous rows in the database. The visibility lookups will be more time consuming, as the chunks could be spread across the entire column in V. However, the visibility times for each chunk $C_n \in F_i$ should also be contigous rows.

5.4 Discussion and concluding remarks

The results of our evaluation indicate the fruitfulness of Passages's chunk based approach in tracing the user's information interaction. Although we did not evaluate Passages in terms of the higher level queries described in previous sections, their effectiveness is implied by the high recall and precision as reported from the granular accuracy analysis of this study. Conversely, file tracing performed relatively poorly. Although file tracing has been successfully used in information retrieval contexts, the accuracy and precision available through Passages makes it not only well suited to improve these systems, but it can also be used in new classes of retrieval problems in which finer detail and stronger accuracy is necessary. That the numbers for paragraph

tracing were also strong indicates that our methods will enable extremely high content precision in tracing, even for small units of information. Additionally, paragraph tracing represents a novel form of tracing for which no comparison exists.

5.4.1 Choice of window size

Window size represents a parameter space which was not empircally tested, but which can be reasoned about. Choice of w involves a trade-off in two dimensions. First, there is a trade off in robustness vs. power. Larger window size creates more robustness; in other words, the likelihood that a positive match is accurate. It also sacrifices power: fewer cases can be reasoned about. For example, consider a window size which is as large as all of the text. Here, a positive match would mean the comparison text contained the entirety of the window, which would be 100% accuracy. However, any text which was slightly different would be a non-match, meaning the window size would be poor for comparing highly similar text. On the other extreme, window size could be reduced to a single character. In this case, comparisons would be much more fine grained, at the expense of accuracy – it reduces to a pure histogram comparison.

The second trade-off is number of chunks generated from text. Larger windows produce fewer chunks, which means fewer database updates during tracing, and fewer database lookups during retrieval. As a permanently increasing dataset, the volume of chunks as a function of window size is an important consideration.

The values used in the Passages experiment were not determined by experimental exploration of the Paramter space. They were chosen to ensure that relatively small regions of text – around 3 sentences — would produce at least a few chunks. This is practically the minimum file size with which we could make confident comparisons.

The combination of two separate indexes partially addressed the lack of empirical assessment of the proper parameters. The index featuring smaller chunks represented

a good compromise over the more efficient (i.e., fewer lookups) but potentially less accurate, larger chunk index. Were results weaker, it would have been more appropriate to explore the parameter space.

5.4.2 Scalability over time

The nature of Passages activity tracing involves a continuously growing database. As discussed previously, there is evidence that even a year's worth of data may be efficiently queried. Query processing power may scale up faster than the growth of data (which should retain linear growth). Still, it is worth considering potential ways to address handling this large dataset over time.

An initial consideration is the objective and technical needs of the particular retrieval application that built on top of Passages's information. For example, an application addressing the problem of finding the "to-read" stack may be able to run the skim-detection queries offline, at periodic intervals, in a manner akin to desktop indexing tools. This would allow the usage summaries of documents to be in cached, quickly retrieval forms. Even more sophisticated applications will likely have some pre-defined use cases – it is unlikely that users will be issuing SQL queries directly. Hence, offline processing may well address potential processing speed issues.

Another approach would be to create alternate databases containing less granular temporal data. The proposed model is capturing activity data on the order of seconds and chunks. Another database may summarize this data to include a chunk's or entire file's activity over the coarse of a minute, hour, or day. This would compress the data significantly. Further, depending on application, it may not compromise quality, especially with much older data where users are less likely to recall temporal information requiring second granularity. For example, although the skim application may need fine data, the other sample queries discussed earlier would likely be of similar quality with

minute or even hour granularity. This optimization, although increasing the size of total data maintained, could dramatically improve the speed of queries which can use less granular data.

5.4.3 Limitations

There are some limitations in our work. The ability to trace the user's interaction with images or video within the UI would make Passages more comprehensive. Although we did observe natural memory decay with our participants, we would like to follow this research with a long-term study in which users' recollection of lost items is weaker, and in which users have their own information need rather than a simulated one.

Cases where files are extremely similar, or have large overlapping regions, could compromise the accuracy of Passages's file activity detection. There are two counterpoints to this problem. First, pure file-tracing could be used in the case of ambiguity to determine which file more recently experienced an access. Second, there is a philosophical question of the nature of tracing. Although the file may not be accurately traced, the information it contains is accurately traced. For example, we may not know which file was actually being used, but we do know which paragraph was being viewed. As file interaction moves toward abstraction, the relevance of the underlying file fades.

While we would have liked to execute a long term study in which users interact with their own files and applications (and may do so in the future), our study focused on accuracy rather than trying to show how certain user tasks may be improved by rich temporal information. We chose our evaluation approach for a number of important reasons. First, while improving the user experience is the primary goal, this cannot be done without first demonstrating the soundness of the underlying methods. Evaluating accuracy requires an oracle, the construction of which is a time consuming process in which a person must manually inspect all instances in which a file is used, which both

infringes on privacy and is prohibitively time-consuming for long term data. Next, as described earlier, research has identified an unsatisfied need for information retrieval systems to support rich temporal representations [5]. Our aim was to fulfill this need and present Passages as a flexible framework over which powerful temporal information management systems could be developed. Not only would Passages directly improve existing systems which use file stream data that is not completely reliable, but we believe it can be easily adapted to new uses by using the primitives in the V and F tables, as well as artifact and temporal retrieval.

5.4.4 Wrap up

This work was in part motivated by our previous experiences with file-based tracing with respect to emerging trends in document interaction. As more computer-mediated work migrates to the web and other non-traditional forms, the number of file types and applications multiplies, the distinction between modalities blurs (e.g., browsing, word processing, and email can all be accomplished via web pages), and information interaction becomes much less structured, making activity tracing greatly more complex. Further, the user's file space is spread out across not only different locations, but different domains of control; where local file tracing can capture the activity of any application, these new remote applications do not operate on local files and hence their activity is not traceable through file based means. Our approach seeks to adapt activity tracing to this new generation of document interaction while still improving upon the current one.

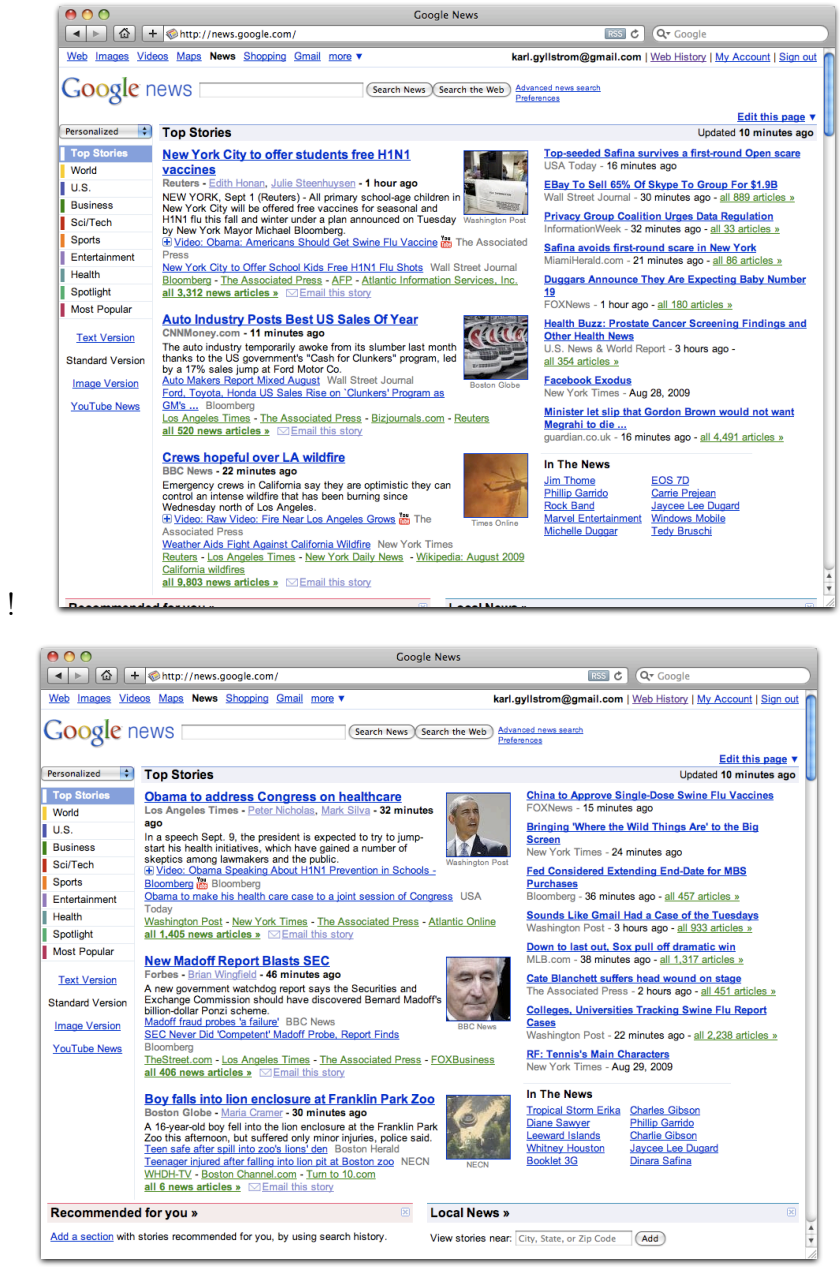


Figure 5.5: *Repeat visits to page.* The contents of the Google News page is depicted at two points in time. Although the browser's history record will indicate two visits to `http://news.google.com`, the visits feature dramatically different page contents. It makes more sense to perceive the events as the viewing of new articles rather than new visits to the same page.

Chapter 6

Concluding remarks

First, we summarize the contributions of each system.

6.1 Summary

6.1.1 Confluence

Confluence improves upon purely file-based tracing approach to context identification by integrating UI-layer tracing. There are some key algorithms within UI-layer tracing which address the causal problems. First, focused task filter (FTF) reduces the noise of unrelated file activity by only considering file events from the currently focused window. It expands upon the time period over which file events can be considered related, better encompassing actual user tasks. Finally, it reduces the importance of files which are used for many disparate tasks, enabling better identification of task relationships. These qualities are independently evaluated and are shown to constitute improvements.

6.1.2 SeeTrieve

SeeTrieve addresses the problems related to the fact that Confluence's tracing is limited to the local filesystem. It does this by abstracting the capture of information to the text which users view in the user interface, and associating this text with files that are accessed at proximal points in time. This enables some forms of non-local information interaction – such as interaction with web pages which are not stored locally – to be captured and used in retrieval.

The use of text is an improvement over Confluence. First, it allows text which does not come from traceable file-sources (e.g., a web email) to still be traced and used in context building. Second, as the tracing is limited to visible text, the context focuses on the information which was actually used (e.g., a particular section of a file). Finally, where Confluence is limited to representing file-to-file relationships, SeeTrieve represents file-to-text relationships, which creates a form of semantic description or tagging. The context-tagging portion, for example, allows words to be associated with files as descriptors. This is impossible in Confluence.

6.1.3 Passages

Passages moves far past Confluence and SeeTrieve to treat visible text as a first class entity, for which it records a history of interaction. There are two areas in which I divide the contributions comprised by this system: First, there are its direct effects on its predecessors, substantially improving upon the accuracy of file tracing. Second, it provides new possibilities in tracing, allowing any viewed text to become a unit of information for which history can be traced. In this sense, Passages renders the traditional file-based view of users' information unnecessary.

6.2 A common thread

The systems described in this thesis reflect an evolution, with each successor addressing limitations in the previous system. In addition, there are ways to configure the systems such that one benefits from the techniques of the other. In this case, the used system acts as a filter or cleaner source of information.

Figure 6.1 depicts one arrangement. In this example, SeeTrieve, which ordinarily monitors file accesses in the filesystem, can benefit from the filtering of Confluence. In this case, SeeTrieve's accuracy will be improved.

Figure 6.2 depicts another arrangement. In this example, SeeTrieve and Confluence draw their file information from Passages rather than directly from the filesystem. This stands to dramatically reduce the noise of filesystem tracing in each system. Furthermore, SeeTrieve still benefits from other aspects of Confluence, such as TaskRank.

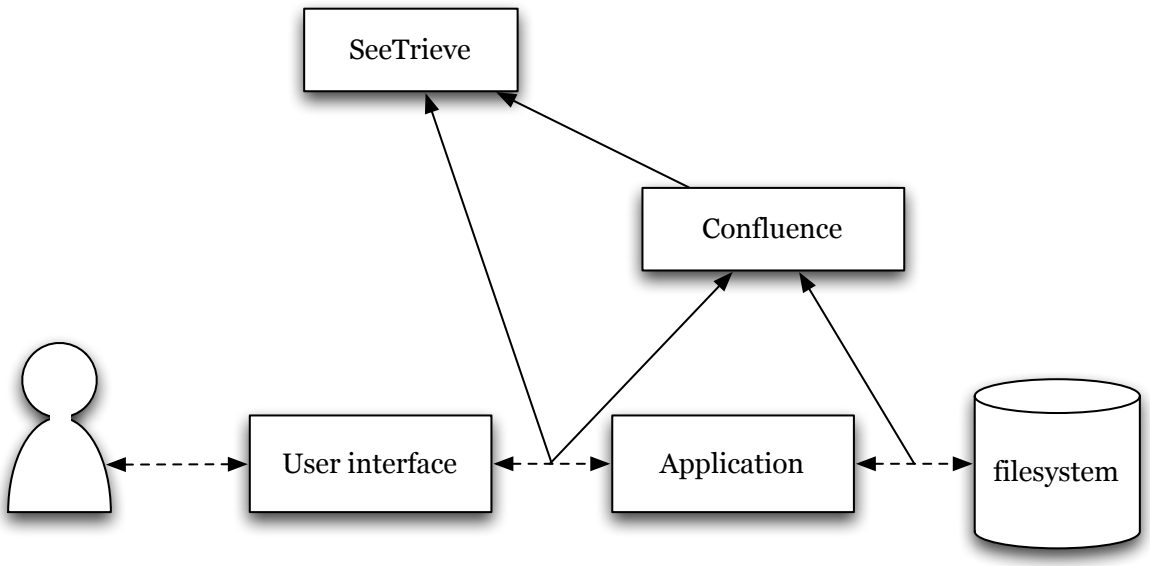


Figure 6.1: Confluence and SeeTrieve

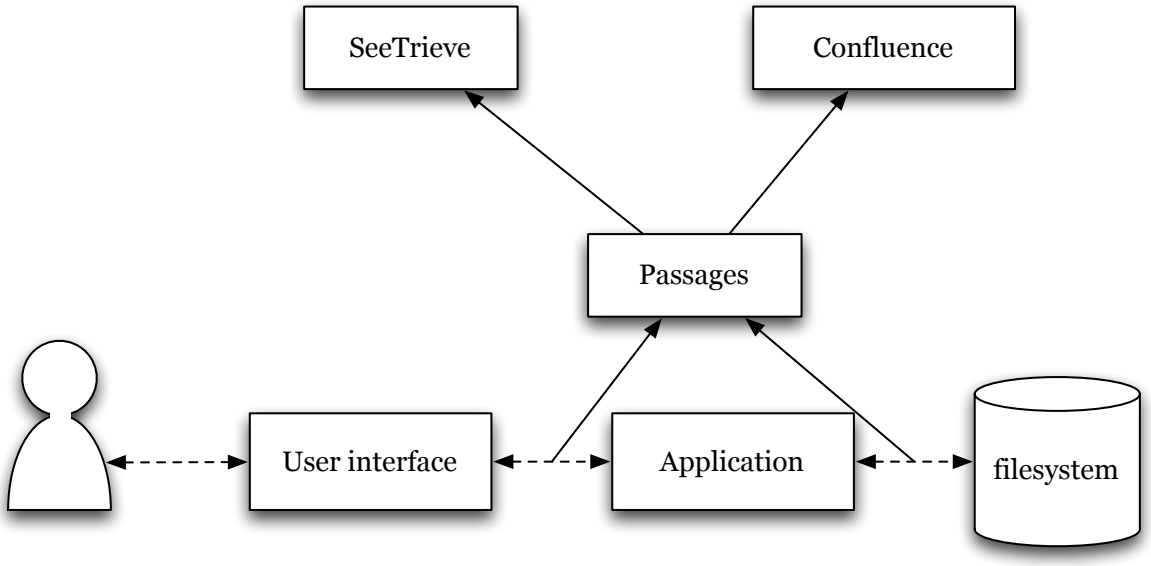


Figure 6.2: Confluence, SeeTrieve and Passages

6.3 Future work and potential applications

6.3.1 User experimentation

The evaluation of personal data sets is a very difficult task [14, 32]. One of the primary trade-offs is in realism vs. the generation of sufficient data points. In order to be natural, we must wait for moments in which a user has a genuine information need. This means that there is less control over the amount of data that can be collected (some participants may take long periods of time before needing to find something). In the case of this thesis, a further constraint is that we can only consider the period over which tracing occurred.

A long term, naturalistic user evaluation of these systems would be useful further work. Particularly, it would be useful to see how these systems operate in the presence of information needs of internal origin (as opposed to requested from the study-giver).

6.3.2 Applications

This section lists possible applications that could use the Passages system to enhance information management for users. The applications in this section are not proposed to be completed within this thesis; rather, they are intended to show how specific information management problems may be easily solved using the Passages framework.

Activity-enhanced ranking

The Passages architecture can be used to improve an existing file search tool by modifying results through information derived from chunk provenance and activity. Activity information can be used as a ranking (i.e., sorting) metric to enhance the traditional retrieval task.

The retrieval system will operate as follows. User queries will be redirected to an existing search tool, such as Google Desktop or Apple's Spotlight. The files in the result pool of the text search tool will have their chunk contents located in the chunk index, and the history of these chunks will be collected and associated with the files of which they are a part.

There are a number of ways in which activity could be used to alter result rankings. First, results which pertain to files that have never actually been accessed by the user can be ignored. For example, if I search with the query "vector", I will get many results pertaining to UNIX system header files. Next, we can allow sorting of results by a function of various temporal criteria, such as recency, frequency, and duration of accesses. Results could be complemented by a visualization of the files accesses over time to provide a trigger for users to remember documents. See Figure 6.3 for a mock-up of such a system.

Finally, the result list can favor a file in which the portion of the file[1] containing the

[1]Because of the chunk-based approach, this sub-file level of granularity is possible.

query has experienced a lot of activity (perhaps relative to the rest of the file). The idea is that activity may be more useful when we only consider the activity on the portion of the file which matches the query. For example, the search query "vector" may match many files which have been active. One of those files, file F_i, may be routinely used, but the portion of F_i containing the term "vector" is rarely viewed. Hence, when a user issues that query, they may not consider F_i to be a document which contains the term (i.e., they were looking for something else). It may be advantageous to rank the result list such that F_i, and files like it, are presented later in the list. This could be additionally useful for within-file searches, where one is trying to find a page within a large document, and may desire to limit results to pages which were more heavily accessed.

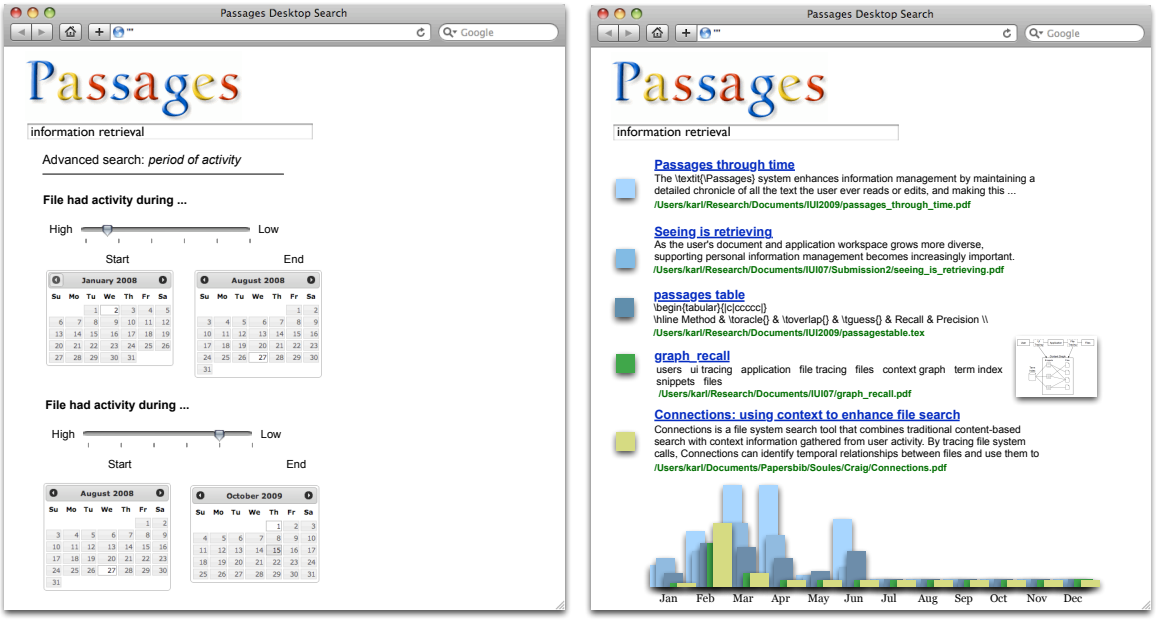


Figure 6.3: Activity-oriented search engine. Above is an example of search engine which allows user to specify activity periods that can be used to filter or rank results.

Active bookmarks/browser history

Browser history is a common way to retrieve previously used web documents. However, browser history is too simplistic to answer certain questions, and the fact that it time stamps documents by initial or most recent access time obscures important aspects of the user's browsing path.

For example, consider a case where a user executes a web search for "context", receives a page of results, and quickly skims through a few of them before finally finding one that satisfies the query, upon which he spends several minutes. If the user later tried to find this page in his history, each page along this path would be represented equally, even though only one of the many pages was actually useful.

Looking at timestamps alone would not suffice to correct this problem. Imagine we used time stamp information to infer the amount of time spent on the page. What would happen if, for example, the user switched to a different tab or application window? The time stamps from different pages would be interleaved even though the user's focus was not.

A further problem occurs when a user frequently accesses a page whose contents vary widely over time. A typical example is a news page, which are updated multiple times a day to reflect breaking news. Over time, many accesses to a page like `http://news.google.com` may accumulate. Even if we could infer time spent reading from inner-access times of pages, we have the problem of not knowing what the contents of the page were when the user read the page.

Fortunately, Passages can easily determine the amount of time spent per page, and we can embed this information into a browser plugin to enhance the existing browser history functionality. For example, to determine a page's history, we simply chunk its contents and use Passages to aggregate, summarize, and report the history of these chunks.

Activity zeitgeist application

Operating system vendors like Apple and Microsoft have done much to facilitate file backup and restoration for consumer PCs, which Apple's *Time Machine* featuring an intuitive interface to browse a system's history. What these systems lack is a way to express the activity of a time period. A date by itself may not be useful; as stated earlier, user's often recall time periods by what happened in them. The granularity in recording user activity within the Passages would enable time periods to be summarized by features like "which documents were read", "how long were they read", and "which portions were read or written".

The zeitgeist application will enable a query specifying date and duration. Activity within this duration will be summarized and presented to the user. Figure 6.4 depicts a mock-up of such a system.

File provenance visualizer

Examining a file by its chunks, we can identify the provenance of each chunk individually. This allows us to recreate the lifetime of a file by its pieces. Questions such as "when was this paragraph read" and "when did this paragraph first appear" could be answered. It would be interesting to build a file editor/viewer which was capable of presenting contents in such a way as to indicate their activity. For example, rendering recently read text as large, opaque, and bold, and older text as smaller and semi-transparent. For another example, a user may be allowed to highlight a given paragraph and query the system history for it. This system achieves a similar effect to that of the prototype described in Hill et al [28], though through a generalized approach rather than a specially designed text application. See Figure 6.5 for a mock-up.

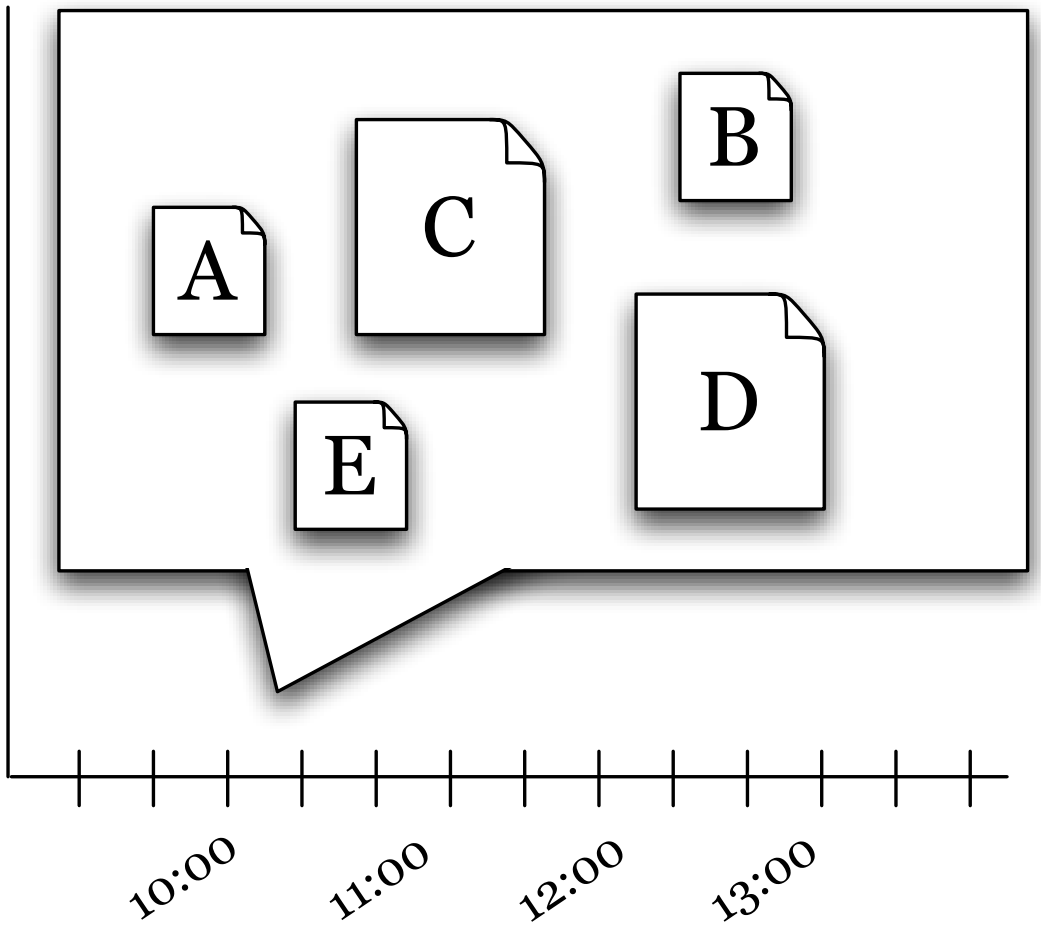


Figure 6.4: Mock-up of a history-viewer application

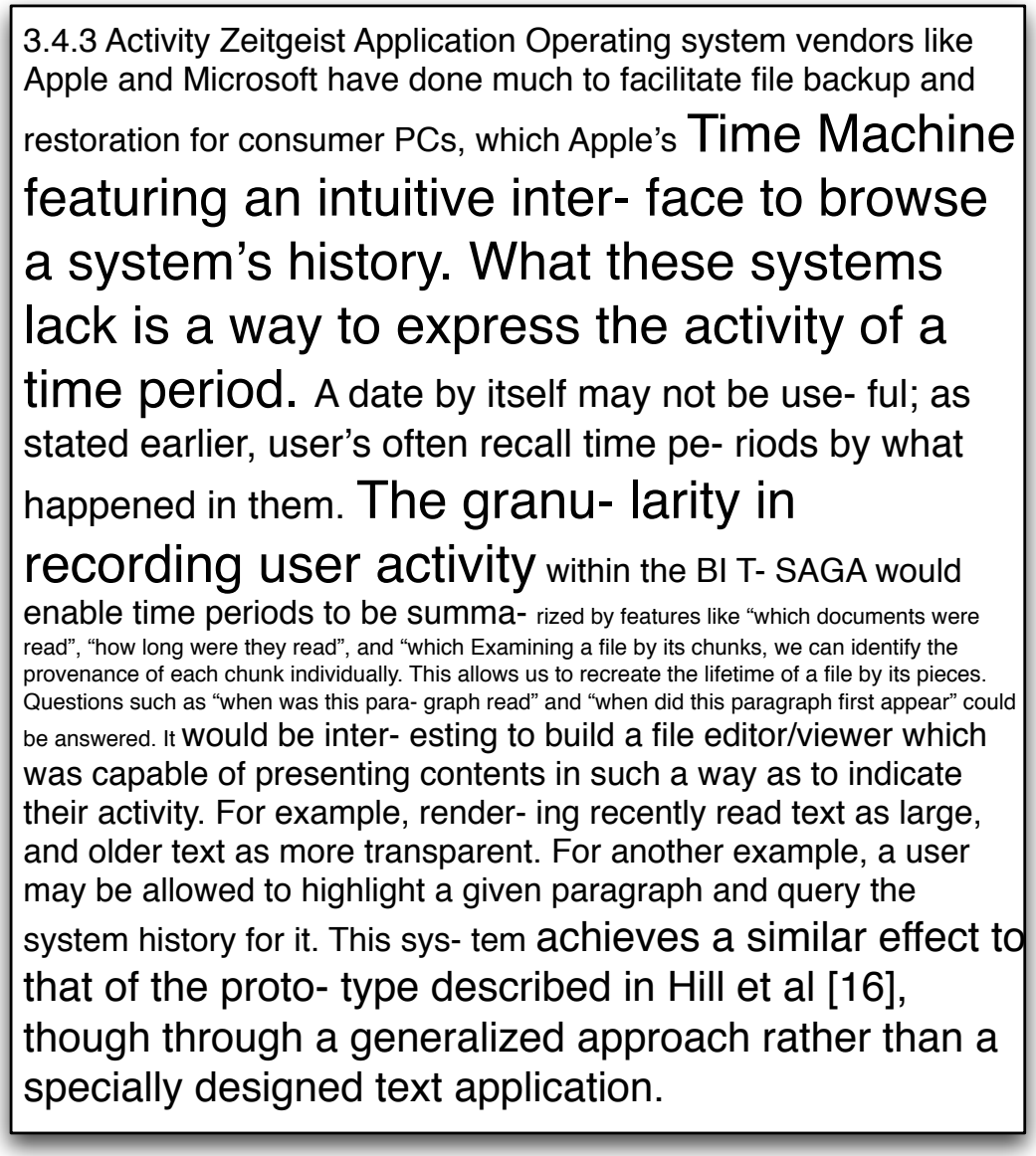


Figure 6.5: Visualizer mock-up.

Pure text retrieval

The text history captured by Passages is not bound to files: it exists independently. Retrieval could operate on text itself. For example, consider the case of a news page in which a snippet is viewed and later, vaguely recalled. Consider the image in Figure 6.6 where we see a snippet about healthcare. The user may have a simple recall desire: "which newspaper was that article from?" which should involve a simple answer. They may issue a browser history query, "healthcare", which returns the page `http://news.google.com`, which, upon re-visiting, contains new articles. The actual answer to the question is no longer contained in the user's personal information workspace. On the other hand, Passages would have kept a record of the text from which the answer could easily have been recalled.

Figure 6.6: A news page at one moment in time. Retrieving information from this page at a later date is impossible because it is constantly changing. Passages would retain a text snapshot, from which later queries could be answered.

Appendix A

Content-based file search

Content-based file search typically comprises an index and a front end that enables users to issue keyword queries (Figure A.1). The index which maps terms to the documents which contain them. The generation of search results from a user query involves returning the documents which contain the terms. Early systems did little beyond this, and how exactly to best do this is a matter of continued research. Identifying files which contain a term is straightforward; one of the primary challenges is *ranking*, or determining how to order results so that higher quality results appear sooner.

Ranking is essential in search engines, as it allows documents which better match the query to be more easy to find (as opposed to an underordered list of all documents containing the term). Quality ranking is one of the qualities which makes Google so effective [1].

One approach which has achieved sustained success is *term frequency-inverse document frequency* (TF-IDF), an intuitive approach to retrieval. The technique is divided between term frequency and inverse document frequency. Term frequency is a mapping between a term T_i and document D_j, specifying the number of occurrences of term T_i in D_j. Conceptually, it represents the importance of a particular term to a particular document. Inverse document frequency (IDF) is a per-term number: it is computed as the log of the number of documents in the index divided by the number of documents in which the term appears. Conceptually, this represents how broad a term is. For

[1] Contrast this aspect to the criteria web search engines used to brag about: number of crawled documents. When an answer can be found in the first page of results, depth is rarely useful.

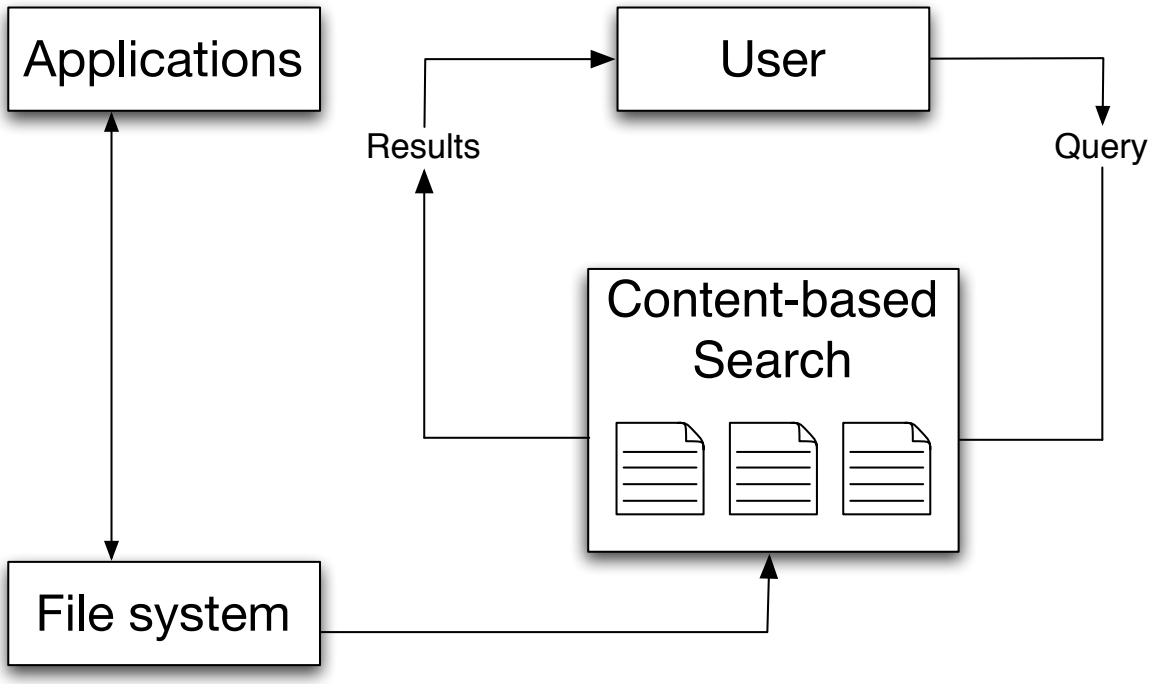


Figure A.1: Traditional contenxt-search architecture.

example, given a corpus containing news articles, a term like "the" will have a low IDF score while a term like "Shakespeare" will have a relatively low IDF score.

The TF-IDF score for a term and document pair is computed by multiplying the TF and IDF scores. This score is used for ranking different documents which contain the term. A high TF-IDF score is one in which the term is relatively important to the document (high TF), while relatively discriminating (high IDF). This is why a query containing the term "Shakespeare" is more likely to generate strong matches than the term "poets"; the former is more discriminating (i.e., narrower).

More sophisticated approaches have been developed. *Language modeling* develops a probabilistic model of the terms in the corpus. Put simply, each document is treated as a language model with some probability of generating the various terms within it. Upon a user query, the query is treated as its own language model and compared to language models in the corpus (i.e., documents) using a statistical model divergence calculation

(e.g., Kullback-Leibler divergence). Documents with lower divergence scores are ranked as stronger matches.

Ranking is difficult in desktop search engines for a number of reasons. First, desktop file spaces lack sophisticated structures in which files relationships to other files are clearly specified via hyperlinks. As stated in Section 2.1, this structure is crucial for web based ranking systems such as PageRank. As a result, desktop search can only use text. As documents are more heterogeneous on desktop file systems than on the web, where documents are designed primarily to be seen by others, text is a difficult dimension to be limited to (e.g., how does one rank a source code file and an email containing the name of a collaborator, based on a query on that person's name?).

Appendix B

Files, Applications, and GUIs

I describe file activity through three layers: filesystems, applications, and graphical user interfaces (GUIs). Although the relationship among these layers is complicated, they are useful abstractions that are practically universal in desktop computing systems.

Filesystems bear the responsibility of storing files, including their content and metadata. File reading and writing is technically more complicated than it appears to the user; for example, write commands, as supported by filesystems, require the caller to specify precise memory addresses to which data is written. Additionally, filesystems do not expose interfaces to the user: file access and manipulation is limited to function calls exposed by the operating system. For this reason, users never directly manipulate files: their access is always moderated by applications.

Applications, then, represent the interface between users and files. Users interaction with applications via the application's user interface, which is almost universally graphical. Interaction occurs through graphical widgets, including windows, menus, buttons, text fields, sliders, and images. The GUI is the medium through which applications represent files, and actions at the GUI are what trigger the application to execute actions, particularly within the filesystem. For example, in a conventional text-editing application, a user executing a "save" command will trigger the application to write the contents of the user's text manipulation to the file with which that user interacts. (See Figure B.1 for a visual of these layers.)

It is possible to record events which occur. The anchor points are where the layers

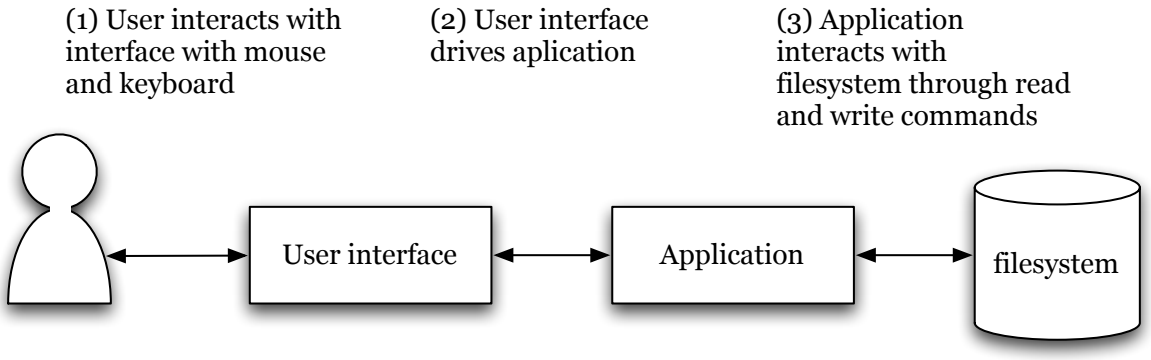


Figure B.1: Layers of interaction.

communicate between one another. Between applications and the filesystem, a set of operating system calls are available. Calls include but are not limited to those described in Figure B.1. Tracing at this layer involves some instrumentation of the operating system, allowing these events to be recorded and their information to be copied and sent to a dedicated file or third party application. Several implementations exist, including Detours (Windows) [29], and dtrace (Unix, Mac OS X) [9]. Although operating systems differ in the manner in which they store and expose interaction upon files, there is enough consistency that tracers for different systems record roughly the same data.

Class	Call	Description	Metadata
Process	Create	[sub]process spawned	time, pid, parent pid
	Destroy	[sub]process spawned	time, pid
File	Read	part of file is read	time, pid, file name
	Write	part of file is written	time, pid, file name
	Open	file is opened for access	time, pid, file name
	Close	file is closed	time, pid, file name

Table B.1: Operating system calls: the interface between application and file system.

Between applications and the GUI, a similar approach is available. Applications must interact with the operating system's implementation of the GUI, and therefore

must call operating system functions. These calls can be traced in a manner similar to filesystems. Figure B.2 depicts some of the calls available.

Class	Call	Description	Metadata
UI	Focus	application window gains focus	time, pid, duration
	Text	text on widget gains visibility	time, pid, duration

Table B.2: Operating system calls: the interface between application and GUI system.

This model presents a number of challenges to systems which attempt to monitor user-file interaction. The systems we present have different ways of dealing with these limitations.

Bibliography

[1] Google desktop. `http://desktop.google.com`.

[2] Indri – language modeling meets inference networks. `http://www.lemurproject.org/indri/`.

[3] T. Bauer and D. Leake. Using document access sequences to recommend customized information. *Intelligent Systems, IEEE see also IEEE Intelligent Systems and Their Applications*, 17(6):27–33, 2002.

[4] Victoria Bellotti and Ian Smith. Informing the design of an information management system with iterative fieldwork. In *DIS '00: Proceedings of the 3rd conference on Designing interactive systems*, pages 227–237, New York, NY, USA, 2000. ACM.

[5] Tristan Blanc-Brude and Dominique L. Scapin. What do people recall about their documents? Implications for desktop search tools. In *IUI '07: Proceedings of the 12th international conference on Intelligent user interfaces*, pages 102–111, New York, NY, USA, 2007. ACM Press.

[6] Richard Boardman and M. Angela Sasse. "stuff goes into the computer and doesn't come out": a cross-tool study of personal information management. In *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 583–590, New York, NY, USA, 2004. ACM.

[7] Jay Budzik and Kristian J. Hammond. User interactions with everyday applications as context for just-in-time information access. In *IUI '00: Proceedings of the 5th international conference on Intelligent user interfaces*, pages 44–51, New York, NY, USA, 2000. ACM Press.

[8] Vannevar Bush. As we may think. *The Atlantic Monthly*, 176(1):101–108, 1945.

[9] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[10] Paul Alexandru Chirita, Claudiu S. Firan, and Wolfgang Nejdl. Personalized query expansion for the web. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 7–14, New York, NY, USA, 2007. ACM Press.

[11] Anton N. Dragunov, Thomas G. Dietterich, Kevin Johnsrude, Matthew McLaughlin, Lida Li, and Jonathan L. Herlocker. Tasktracer: a desktop environment to support multi-tasking knowledge workers. In *IUI '05: Proceedings of the 10th international conference on Intelligent user interfaces*, pages 75–82, New York, NY, USA, 2005. ACM Press.

[12] Nicolas Ducheneaut and Victoria Bellotti. E-mail as habitat: an exploration of embedded personal information management. *Interactions*, 8(5):30–38, 2001.

[13] Susan Dumais, Edward Cutrell, Raman Sarin, and Eric Horvitz. Implicit queries (iq) for contextualized search. In *SIGIR '04: Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 594–594, New York, NY, USA, 2004. ACM.

[14] David Elsweiler and Ian Ruthven. Towards task-based personal information management evaluations. In *SIGIR '07: Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 23–30, New York, NY, USA, 2007. ACM Press.

[15] David Elsweiler, Ian Ruthven, and Christopher Jones. Towards memory supporting personal information management tools. *Journal of the American Society for Information Science and Technology*, 58(7):924–946, 2007.

[16] Kave Eshghi and Hsiu Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. Technical Report 30, Hewlett-Packard Labs, 2005.

[17] Scott Fertig, Eric Freeman, and David Gelernter. Lifestreams: an alternative to the desktop metaphor. In *CHI '96: Conference companion on Human factors in computing systems*, pages 410–411, New York, NY, USA, 1996. ACM Press.

[18] George Forman, Kave Eshghi, and Stephane Chiocchetti. Finding similar files in large document repositories. In *KDD '05: Proceeding of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 394–400, New York, NY, USA, 2005. ACM.

[19] Xiaobin Fu, Jay Budzik, and Kristian J. Hammond. Mining navigation history for recommendation. In *IUI '00: Proceedings of the 5th international conference on Intelligent user interfaces*, pages 106–112, New York, NY, USA, 2000. ACM Press.

[20] Jim Gemmell, Gordon Bell, Roger Lueder, Steven Drucker, and Curtis Wong. MyLifeBits: fulfilling the Memex vision. In *MULTIMEDIA '02: Proceedings of the tenth ACM international conference on Multimedia*, pages 235–238, New York, NY, USA, 2002. ACM.

[21] Daniel Gonçalves. Telling stories about documents. Technical report, INESC-ID, 2003.

[22] Daniel Gonçalves. Real stories about real documents: Evaluating the trustworthiness of document-describing stories. Technical report, INESC-ID, 2005.

[23] Daniel Gonçalves and Joaquim A. Jorge. Describing documents: what can users tell us? In *IUI '04: Proceedings of the 9th international conference on Intelligent user interfaces*, pages 247–249, New York, NY, USA, 2004. ACM Press.

[24] Daniel Gonçalves and Joaquim A. Jorge. In search of personal information: Narrative-based interfaces. In *IUI '08: Proceedings of the 13th international conference on Intelligent user interfaces*, 2008.

[25] Daniel Gonçalves and Joaquim A. Jorge. In search of personal information: narrative-based interfaces. In *IUI '08: Proceedings of the 13th international conference on Intelligent user interfaces*, pages 179–188, New York, NY, USA, 2008. ACM.

[26] Karl Gyllstrom and Craig Soules. Seeing is retrieving: building information context from what the user sees. In *IUI '08: Proceedings of the 13th international conference on Intelligent user interfaces*, pages 189–198, New York, NY, USA, 2008. ACM.

[27] Karl Gyllstrom, Craig Soules, and Alistair Veitch. Activity put in context: identifying implicit task context within the user's document interaction. In *IIiX '08: Proceedings of the second international symposium on Information interaction in context*, pages 51–56, New York, NY, USA, 2008. ACM.

[28] William C. Hill, James D. Hollan, Dave Wroblewski, and Tim McCandless. Edit wear and read wear. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 3–9, New York, NY, USA, 1992. ACM Press.

[29] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, Seattle, WA, July 1999. USENIX.

[30] Victor Kaptelinin. UMEA: translating interaction histories into project contexts. In *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 353–360, New York, NY, USA, 2003. ACM Press.

[31] David Karger, Karun Bakshi, David Huynh, Dennis Quan, and Vineet Sinha. Haystack: A general purpose information management tool for end users of semistructured data. In *CIDR*, pages 13–26, 2005.

[32] Diane Kelly. Evaluating personal information management behaviors and tools. *Commun. ACM*, 49(1):84–86, 2006.

[33] Alison Kidd. The marks are on the knowledge worker. In *CHI '94: Conference companion on Human factors in computing systems*, page 212, New York, NY, USA, 1994. ACM.

[34] Aparna Krishnan and Steve Jones. Timespace: activity-based temporal visualisation of personal information spaces. *Personal Ubiquitous Comput.*, 9(1):46–65, 2005.

[35] B. Kwasnik. How a personal document's intended use or purpose affects its classification in an office. *SIGIR Forum*, 23(SI):207–210, 1989.

[36] Oren Laadan, Ricardo A. Baratto, Dan B. Phung, Shaya Potter, and Jason Nieh. DejaView: a personal virtual computer recorder. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 279–292, New York, NY, USA, 2007. ACM.

[37] M. Lansdale. The psychology of personal information management. *Applied Ergonomics*, 19(1):55–66, March 1988.

[38] Wendy E. Mackay. More than just a communication system: diversity in the use of electronic mail. In *CSCW '88: Proceedings of the 1988 ACM conference on Computer-supported cooperative work*, pages 344–353, New York, NY, USA, 1988. ACM.

[39] Thomas W. Malone. How do people organize their desks? implications for the design of office information systems. *ACM Trans. Inf. Syst.*, 1(1):99–112, 1983.

[40] Richard Mander, Gitta Salomon, and Yin Yin Wong. A "pile" metaphor for supporting casual organization of information. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 627–634, New York, NY, USA, 1992. ACM.

[41] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. Provenance-aware storage systems. In *In proceedings of the 2006 USENIX Annual Technical Conference*, 2006.

[42] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 174–187, New York, NY, USA, 2001. ACM.

[43] Nuria Oliver, Greg Smith, Chintan Thakkar, and Arun C. Surendran. SWISH: semantic analysis of window titles and switching history. In *IUI '06: Proceedings of the 11th international conference on Intelligent user interfaces*, pages 194–201, New York, NY, USA, 2006. ACM Press.

[44] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[45] Elin Rønnby Pedersen and David W. McDonald. Relating documents via user activity: the missing link. In *IUI '08: Proceedings of the 13th international conference on Intelligent user interfaces*, pages 389–392, New York, NY, USA, 2008. ACM.

[46] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, Cambridge Mass., 1981.

[47] Tye Rattenbury and John Canny. CAAD: an automatic task support system. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 687–696, New York, NY, USA, 2007. ACM Press.

[48] Pamela Ravasio, Sissel Guttormsen Schär, and Helmut Krueger. In pursuit of desktop evolution: User problems and practices with modern desktop systems. *ACM Trans. Comput.-Hum. Interact.*, 11(2):156–180, 2004.

[49] Jun Rekimoto. Time-machine computing: a time-centric approach for the information environment. In *UIST '99: Proceedings of the 12th annual ACM symposium on User interface software and technology*, pages 45–54, New York, NY, USA, 1999. ACM.

[50] B. Rhodes and T. Starrier. The remembrance agent: A continuously running automated information retrieval system. In *The Proceedings of PAAM 96*, pages 487–495, London, UK, April 1996.

[51] Bradley J. Rhodes. Margin notes: building a contextually aware associative memory. In *IUI '00: Proceedings of the 5th international conference on Intelligent user interfaces*, pages 219–224, New York, NY, USA, 2000. ACM.

[52] Jianqiang Shen, Lida Li, Thomas G. Dietterich, and Jonathan L. Herlocker. A hybrid learning system for recognizing user tasks from desktop activities and email messages. In *IUI '06: Proceedings of the 11th international conference on Intelligent user interfaces*, pages 86–92, New York, NY, USA, 2006. ACM Press.

[53] Craig A. N. Soules and Gregory R. Ganger. Connections: using context to enhance file search. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 119–132, New York, NY, USA, 2005. ACM Press.

[54] Kazunari Sugiyama, Kenji Hatano, and Masatoshi Yoshikawa. Adaptive web search based on user profile constructed without any effort from users. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 675–684, New York, NY, USA, 2004. ACM Press.

[55] Alan Wexelblat and Pattie Maes. Footprints: history-rich tools for information foraging. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 270–277, New York, NY, USA, 1999. ACM.

[56] Steve Whittaker and Candace Sidner. Email overload: exploring personal information management of email. In *CHI '96: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 276–283, New York, NY, USA, 1996. ACM Press.