Tracing, indexing, and retrieving desktop activity with the Passages system

Karl Gyllstrom
The University of North Carolina at
Chapel Hill
karl@cs.unc.edu

ABSTRACT
Search has come to dominate the way we manage information, so it is useful to make new forms of information amenable to retrieval. Our interaction with our desktop takes place through applications' user interfaces, which convey large amounts of information to us through text. Much of this text is a crucial part of our tasks, often displaying the contents of files, emails, and web pages with which we interact. By recording and indexing it, we can transform desktop activity into a useful, persistent, and retrievable form.

The Passages system uses this text to build a rich representation of the user's interaction history with their desktop and associate this history with their document and information workspace. As time is an important attribute in users' recall of their personal information, this association introduces novel possibilities for information retrieval.

ACM Classification: H.4.2 [User Interfaces]: Graphical user interfaces (GUI)

General terms: Design, Experimentation, Human Factors

Keywords: time-machine computing, context, information retrieval

INTRODUCTION
The typical user interacts with their desktop primarily through applications' graphical user interfaces, through which large amounts of textual data are presented to the user. This text can be captured and cheaply stored, making it amenable to indexing and retrieval. Conceptually, this means all the text with which a user interacts is persistent and can be recalled.

The text made visible through the UI — which I will refer to as the text stream — offers unique advantages to information management and retrieval. For one, the text stream contains rich temporal information about its use, as it can be known when a given text-bearing window is focused and when it loses focus [7]. This allows it to determined, for example, the duration over which a given sequence of text (e.g., a paragraph) was visible to the user.

Since applications often display the contents of files (e.g., a PDF), the text stream is partially composed of those files' text, allowing the user's personal files to inherit this temporal information. For example, we could determine exactly when and for how long a given file or file portion has been read by the user, enabling queries such as "what times this week did I work on this section of the draft?" This granularity of access information is unavailable to traditional operating systems, which, due to the decoupling between applications and the filesystem, can only coarsely approximate this information [7, 8].

Much of the text displayed in the UI does not come from traditional file sources, and hence is not recallable through typical search systems. For example, within the web are many dynamic pages containing rapidly updating information (e.g., news aggregators) whose form is too fluid for traditional indexing (e.g., through a history log). A user trying to recall information from one of these pages may access their history log only to find the information has since changed. Hence, the text stream may be able to address information needs that desktop search tools cannot by retrieving information that has either changed since it was viewed or never existed in the user's filesystem.

Despite these opportunities, there are a number of important challenges in effectively indexing and making sense of the user's interaction with UI text, for which I designed the *Passages* system. Passages captures the user's text stream and transforms it into a rich, granular, information-interaction history. Since interaction history is an attribute that users often employ in conceptually recalling documents, supporting it can improve information management and retrieval [2, 3, 6, 10]. For example, Passages is designed to answer questions like, "which of these conference papers have I not yet read", "which documents did I read when writing this literature review section", "what files was I working on before I committed this code", and "which documents did I spend the most time on last month?"

PASSAGES
Passages is not a complete information management system, but rather a framework which traces the user's activity, analyzes and indexes that activity, and exposes temporal information about this activity to other applications (e.g., a file browser or search tool). Passages supports two main retrieval modes: artifact and temporal. Artifact retrieval answers requests for the temporal history of a unit of information. 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 objects

UIST'08, October 19–22, 2008, Monterey, CA, USA
ACM 978-1-59593-975-3/08/10

(e.g., files) which experienced some activity within this period. An example temporal query would be, "what files did I work on this morning?" Complex queries can be formed by combining these modes. For example, one may wish to determine which papers they have downloaded in the last month (temporal) which were skimmed but not completed, where "skimmed" could be defined as having been read for under 5 minutes or having not been read in its entirety (artifact).

Passages comprises two subsystems. The *tracing* system incorporates two system event streams to identify and index the user's activity. The *retrieval* system allows high level queries about activity using this index of information.

Tracing

The tracing system records event streams from two system sources: the user interface (GUI) and the filesystem. Tracing user behavior at the UI 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 within its visible region, or by minimizing a previously active window. The text contents of the window — which I 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 owning application. If some small duration transpires with no new focus event, the tracing system acquires a new snippet; this handles cases where a given application window gains new content (e.g., a browser surfing to a new web page). Note that duration is implicit in the snippet stream; the duration over which a text snippet was visible is the difference between time stamps of that snippet and its subsequent snippet.

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 time at which they occur.

The tracing system is purposefully low-level in order to avoid application specific design, which would require retrofitting countless applications to be time aware. As a consequence of this design decision, the text stream is undifferentiated, redundant, and difficult to reason about. Consider the problem of implementing artifact and temporal retrieval modes with this stream. To determine for how long a given file F was active, we would first have to search for all snippets sharing some content overlap with F (using a *diff* based comparison), then tally the timing information for each of those snippets. To determine which files experienced activity during time range T, we would have to find all snippets created during the range, then search the entirety of the user's filesystem for files which shared content overlap with any of the snippets. Clearly, neither of these approaches is tractable for user activity, which is a permanently growing dataset.

Conceptually, making sense of the text stream is like a computer vision problem: given a raw 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 object). Our approach is to break the snippets into small, atomic text units, which each feature a unique identifier based on their contents. I accomplish this through *landmark chunking*, a process which breaks a sequence of text into smaller subsequences, or chunks [4]. Using parameters D and r, where $r < D$, a fixed width sliding window of width W is run across the file contents, character by character. At every position k, an efficient fingerprint algorithm is applied to the contents of the window to achieve a value F_k. If $F_k \ mod \ D = r$, position k represents a chunk boundary.

A unique and important effect 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 the 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).

At the point in which a snippet is created, Passages breaks the contents of each snippet into chunks, whose contents are individually processed with the MD5 hash function to attribute a unique, small (128 bits) identifier to them. Conceptually, this allows for a transformation in representing a snippet from a continuous sequence of text to a set of hash values[1]. The chunks derived from a snippet inherit the timing information from the snippet, creating a small set of 3-tuples of the form (H, S, E), where H is the hash value, and S and E are the points at which the hash gained and lost visibility, respectively.

The filesystem and file event stream are treated as follows. Initially, all of the user's files are chunked and hashed[2]. 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 4-tuples of the form (H, F, S, E), 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 hash first appeared within the file, and E is the time at which (if ever) the hash is no longer present within the file.

Retrieval

The tuples defined in the previous section are entered into two tables — visibility table V and file table F — within a relational database. For a given hash, 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, it is simple to uncover the provenance, lineage, and activity of a chunk of text as it exists within both layers.

This becomes useful when synthesizing information from multiple related chunks. For artifact queries, where activ-

[1]This form is useful in tasks such as efficiently approximating document content similarity, typically called *document fingerprinting*.

[2]Some application-specific plugins (e.g., PDF) are used to maximize coverage

ity of a file is identified, Passages first identifies the chunks pertaining to that file in F, looks up the individual activity histories of each chunk in V, then combines and summarizes these histories. Note that because chunks are granular, this information reveals more than when a file was read: it can reveal activity information about individual parts of the file.

To implement temporal queries, where the file activity which occurred within a given time range is identified, Passages queries V for all chunks which experienced activity within the time range, and then queries F for all files which contain a large number of those chunks. Because date ranges are arbitrary, temporal queries scale from short (e.g., seconds) to large (e.g., months) durations.

Information on chunks 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[3].

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 have locations within the filesystem. Here I justify their use by relating the theoretical properties of landmark chunking to the system objectives.

Detection in the UI

Given snippet T_i containing — in part — $text_i$ which itself generates chunk set C of which chunk C_i is a part, it is highly likely that occurrences of $text_i$ in other snippets will still yield chunk C_i when processed with landmark chunking. The significance of this lies within the chaotic relationship between a document and the way in which its contents are displayed through the application. For example, at two different times, the user scrolls to overlapping positions P_n and P_{n+1} of a document displayed by their application, generating snippets S_n and S_{n+1}. If these snippets were chunked using a fixed width window, the chunks produced by the two snippets would likely have no set overlap despite the content overlap of the snippets. However, with landmark chunking, the set of chunks pertaining to the portion of P_n and P_{n+1} that overlap will be largely the same between the snippets' respective chunk sets.

Another example is when a document is presented to the user alongside surrounding textual noise. For example, when a user opens a web-based mail through the browser, typically the text contents displayed include information that is peripheral to the email, such as advertisements, banners, and menus. Because landmark chunking creates boundaries using local contents, most of the chunks (typically, all but the first) of the email will not be affected by this peripheral text. Hence, even in subsequent viewings, where the advertisements are different, or if the viewing is through a local mail client without any peripheral text at all, the chunks pertaining to the email itself will remain the same.

The importance of this property is that whenever a set of text (e.g., a paragraph) is viewed in the UI, the chunks it produces will be largely the same as all previous viewings of that text. Hence, activity upon that text will nearly always be measurable whenever it is viewed, providing reliability and comprehensiveness to this approach.

Detection in the Filesystem

Revisiting the overlap problem described above, imagine that instead of comparing two overlapping snippets, we compare a snippet containing a portion of a given file (i.e., scrolled to an arbitrary position) to the contents of the file itself. As in the previous example, the chunks pertaining to the overlapping region will be largely the same between snippet and file. Hence, if a given chunk exists in a file, it is likely that whenever the portion of that file containing the chunk is displayed in the UI, the chunk will be detected. This unifies information between the UI and filesystem spaces.

RELATED WORK AND REMAINING PROBLEMS

Many modern systems and applications have rudimentary support for time, although it is typically limited to qualities such as the document's creation or most recent edit date. These qualities coarsely approximate documents' usage, often to the extent of conveying a misleading representation of the user's true interaction with them.

For example, web browsers tend to maintain history logs which contain browsable records of the pages users previously accessed, as well as the times of access. Consider the case where the user undertakes a web search on HCI conferences, first visiting site A, which the user determines to be unhelpful, then visiting sites B and C. After determining C to be unhelpful too, the user goes back in their browser history to B, which they read for the next 20 minutes. As they read it, they occasionally refer to another browser tab displaying site E, a sports site displaying live results, which the user opened previously to the web search. Since the browser's history record is based on access time — or the time in which the page is originally loaded — the timeline it conveys does not faithfully represent the user's true interaction with the web pages. Namely, site E will appear first even though it was used concurrently with site B, and C will appear after B even though the user spent the majority of their time on B after visiting C. Further, each item will have equal representation within the history, even though some sites were used heavily while others were only glanced at.

In operating systems' filesystems, the interaction information for a document is typically limited to creation date and last edit (with application cooperation, last opening time can be recorded). This limitation is fundamentally a consequence of the decoupling between applications — through which user-document interaction occurs — and the filesystem — where this interaction ultimately manifests. Tracing events which occur on files through the filesystem is one approach to capturing file interaction history, though, as our previous

[3]This could be combined with our previous work to enable the retrieval of non-file information [7].

research shows, limiting observation of activity to the filesystem alone is problematic both in terms of signal and noise [8]. With respect to signal, much activity is not recorded; for example, when a user opens a PDF file through an application, that application may read the file to memory and not access it again, even though the user still interacts with the PDF through the application. Noise appears when background tasks, such as virus scanners, generate file activity which is mistakenly attributed to the user.

These examples serve to detail instances of a broader challenge in tracing document activity; much of the problem is due to the limited perspective of a file-based approach, where only access or creation events can be detected and other information is obscured. This approach is prevalent in temporal systems, from email, to filesystems, to the web. The user interface offers a strong advantage over the file based approach adopted by the previous examples, since it is — by definition — more tightly coupled with the user's actions and hence more informative about their activity. In the problem examples, this limitation is removed by following activity in the UI. For example, in the browser case, identifying which page the user actually looks at is more informative than the times in which those pages are first accessed by the browser (the file example is illustrated in [8]). However, linking the UI to file activity is not trivial; Passages provides a novel way to establish this link.

Some research systems have adopted novel approaches to supporting time-based file interaction. The *LifeStreams* system described by Fertig et al. replaces the traditional filesystem of hierarchies with a filesystem represented entirely by a stream of documents ordered by creation or access time [5]. Unlike Passages, *LifeStreams* is a dramatic overhaul of the current desktop paradigm rather than an adaptation of existing systems to better support time.

Edit Wear/Read Wear, described by Hill et al. is a text editor instrumented to capture and display the degree of reading and editing that individually occurs upon each line within a file [9]. Rekimoto's *Time-Machine Computing* augments the user's filesystem with a temporal visualization, using information gleaned through a limited form of activity tracing (e.g., application hooks to ascertain certain forms of file activity) [12]. These systems reflect the range that Passages was designed to support, from the sub-document granularity of *Edit Wear/Read Wear* to the entire workspace of *Time-Machine-Computing*. Both of these systems could either be implemented using or improved by the rich information captured and exposed by Passages.

The *MyLifeBits* project aims to support lifetime personal information stores by retaining and managing as much of the user's personal information as possible [1]. Several systems reveal task-oriented file relationships among files by tracing their times of access and identifying temporal commonalities [7, 8, 11, 13]. These systems would benefit from Passages's refined file activity information.

Desktop search tools like *Google Desktop* or Apple's *Spotlight* enable users to recall documents through content-based queries. These tools can be enhanced using Passages to more highly rank files that are more active or whose portions matching the query are more active, or prune results of files never used. Additionally, Passages could return retrieve information from the text stream itself, in cases where the user is trying to recall something they saw rather than something existing in one of their files.

CONCLUDING REMARKS

By recording and indexing the user's interaction with text within the UI, Passages offers novel ways to support granular, temporal queries upon the user's information space which can enable new, useful forms of information retrieval.

REFERENCES

1. G. Bell and J. Gemmell. A digital life. *Scientific American*, March 2007.
2. T. Blanc-Brude and D. L. Scapin. What do people recall about their documents? Implications for desktop search tools. In *IUI '07*, pages 102–111, New York, NY, USA, 2007. ACM.
3. D. Elsweiler, I. Ruthven, and C. Jones. Towards memory supporting personal information management tools. *JASIST*, 58(7):924–946, 2007.
4. K. Eshghi and H. K. Tang. A framework for analyzing and improving content-based chunking algorithms. Technical Report 30, Hewlett-Packard Labs, 2005.
5. S. Fertig, E. Freeman, and D. Gelernter. Lifestreams: an alternative to the desktop metaphor. In *CHI '96*, pages 410–411, New York, NY, USA, 1996. ACM.
6. D. Gonçalves and J. A. Jorge. In search of personal information: narrative-based interfaces. In *IUI '08*, pages 179–188, New York, NY, USA, 2008. ACM.
7. K. Gyllstrom and C. Soules. Seeing is retrieving: building information context from what the user sees. In *IUI '08*, pages 189–198, New York, NY, USA, 2008. ACM.
8. K. Gyllstrom, C. Soules, and A. Veitch. Confluence: enhancing contextual desktop search. In *SIGIR '07*, pages 717–718, New York, NY, USA, 2007. ACM.
9. W. C. Hill, J. D. Hollan, D. Wroblewski, and T. McCandless. Edit wear and read wear. In *CHI '92*, pages 3–9, New York, NY, USA, 1992. ACM.
10. B. Kwasnik. How a personal document's intended use or purpose affects its classification in an office. *SIGIR Forum*, 23(SI):207–210, 1989.
11. T. Rattenbury and J. Canny. CAAD: an automatic task support system. In *CHI '07*, pages 687–696, New York, NY, USA, 2007. ACM.
12. J. Rekimoto. Time-machine computing: a time-centric approach for the information environment. In *UIST '99*, pages 45–54, New York, NY, USA, 1999. ACM.
13. C. A. N. Soules and G. R. Ganger. Connections: using context to enhance file search. In *SOSP '05*, pages 119–132, New York, NY, USA, 2005. ACM .