Activity put in context: Identifying implicit task context within the user's document interaction

Karl Gyllstrom
UNC-Chapel Hill / HP Labs
karl@cs.unc.edu

Craig Soules
HP Labs
craig.soules@hp.com

Alistair Veitch
HP Labs
alistair.veitch@hp.com

ABSTRACT

Modern desktop search is ill-fitted to our personal document workspace. On one hand, many of the methods which render web search effective cannot be applied on the desktop. On the other, desktop search does not take full advantage of attributes that are unique to our personal documents. In this work, we present Confluence, a desktop search system that addresses this problem by capturing the task context within which a user interacts with their documents. This context is then integrated with traditional desktop search techniques to enable task-based document retrieval.

Building upon Connections, a system that identifies task context by passively monitoring the user's interaction with their documents within the file system, Confluence also traces user activity within the user interface and incorporates methods to analyze and integrate this new stream of information. We show that this approach significantly improves the accuracy of task identification, achieving 25% to 30% better recall.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Clustering

General Terms

Human Factors, Experimentation

Keywords

Contextual search

1. INTRODUCTION

Personal document management is difficult. Due to the inherent challenges in manual organization, search is an appealing approach to document management, as it is generally fast and allows users to defer or bypass placing documents into rigid hierarchies. However, due to large structural differences between the web and one's personal document space, desktop search performs comparatively worse to web search in retrieval tasks [12]. On the web, hyperlinks between documents describe their contextual relationships, providing the foundation for structural search algorithms like PageRank as well as techniques for classifying non-textual data. This rich, meaningful structure is built collectively by the world community of web authors and users; when we issue a successful web search, then, we are in essence leveraging the organizational work of those who came before us. As a personal data set, only users can meaningfully organize their personal document workspaces; a task they are loathe to do because the cognitive load required in such complex organization is often prohibitively high [7, 10]. Hence, the difference in quality between local and web searches is inherent in the separation of the personal and public document spaces, and will likely persist even in the face of better system support for manual organization.

On the other hand, there is an important attribute (among many others) unique to personal data that is not effectively used by desktop search systems: task context. In attempts to recall a given document, users often remember other documents which were used with it as part of a common task [1, 4]. Hence, search systems would be improved by supporting the retrieval of misplaced documents by contextually related files which are better remembered — a form of task-based retrieval. What is needed is accurate techniques for identifying and applying contextual relationships among personal documents in the absence of user supervision or intervention.

Temporal locality, the idea that events which occur at closer points in time are more likely to be conceptually related, has been applied with success in numerous settings involving user activity [8, 9, 12]. Specifically, file accesses by the user have been shown to exhibit this property, allowing the inner-file relationships to be identified independently of their contents [12]. For example, if we see someone read three PDF files in a row, we can infer a likely relationship among those files without knowledge of their contents or the purpose of the task (e.g., literature review, assignment grading).

Unfortunately, by monitoring the user's activity within the filesystem, we observe that the abstraction of file interaction by modern applications has created a divergence between the way users interact with their documents and the way that interaction manifests on the filesystem. This divergence limits a purely filesystem based perspective. On the other hand, 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. The challenge, then, is to monitor the user at a low enough level to obviate application support while accounting for the ways in which information at this level fails to accurately reflect the user's activity. Overcoming this file interaction abstraction is essential for any system attempting to deeply support activity context on the desktop.

To address this challenge, we designed Confluence, a context-enhanced document retrieval system that builds information about

IIiX'08, Information Interaction in Context, 2008, London, UK.

the user's activity by events from the filesystem and graphical user interface. These two event sources provide complementary information which allows us to address the file abstraction problem without losing generality through application-specific design. This work builds on Connections, a context-enhanced retrieval system that uses inter-file relationships identified by access patterns to improve both the recall and precision of personal file system search over traditional content-only means [12]. By adding UI layer events, Confluence improves the quality of these relationships, further improving retrieval. 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 between 25% and 30% more correct files at 30 results than Connections.

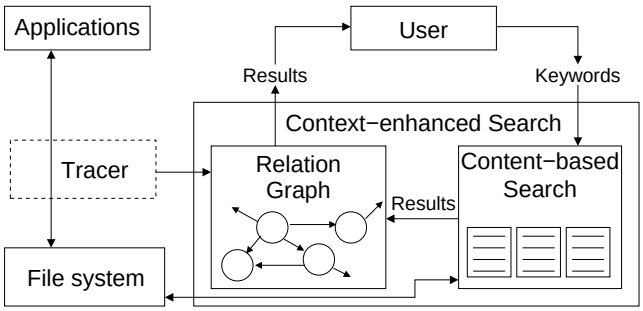

Figure 1: Connections framework.

2. BACKGROUND

Multiple systems adopt the approach of viewing the user's document workspace from a task perspective. UMEA [5] and TaskTracer [2] organize data according to discrete tasks that the user explicitly defines, and monitor various events which are then associated with the task identified by the user as currently active. Haystack [6] enables users to collect disparate data objects (e.g., emails, web pages, personal documents) into a single contextual grouping; while very flexible and using a more fine-grained information unit than the document, this system requires application specific adapters. The limitation in these systems is that (a) they require the user to explicitly organize their data or identify tasks, and (b) they require some level of application support.

Some systems have approached task support by identifying data relationships from the user's activity. 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 [11]. CAAD uses file access activity to inform a task visualizer interface, but the tracing approach is somewhat application specific (e.g., relying on methods such as file-locks which are not universally adopted by applications) — further, the accuracy of task identification is not rigorously evaluated.

3. 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 a large source of noise. 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. There are two primary problems that we address within this disconnect.

Background applications: While the user's current task generates a stream of activity within the file system, a large number of user and system 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 the file events of the user's current task, making it difficult to identify the events of user origin. As we describe in Section 5.1, this problem is enormous; our experiment shows that only 4% of the file event stream corresponds to files with which the user directly interacts.

Hidden activity: Active user applications can also generate file events without application layer activity. Many applications employ numerous configuration and state maintenance files which are invisible or unavailable to the user but necessary for the process's 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 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 from the file layer both sensitive and brittle.

On the other hand, the array 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. This approach introduces unique problems that require new methods to solve. Rather than mitigate the noise by adapting our tracing system to a less informative layer (e.g., the application), we introduce an additional stream of information — the user interface event stream — to complement and inform the analysis of the user's file activity.

4. CONFLUENCE

Confluence builds on the existing Connections framework by introducing novel algorithms to integrate event streams from two distinct sources: the user interface and the file system. This section overviews the relevant components of the Connections framework within which Confluence operates (Section 4.1) and describes the new algorithms required to enable Confluence (Sections 4.2 and 4.3).

4.1 Connections and Context Building

Figure 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 [12]. 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.

4.2 Focused task filtering

Confluence traces the user's interaction with *application windows*, specifically, when the user switches focus between windows (e.g., by clicking on a window to bring it to the forefront). The Focused Task Filtering (FTF) algorithm approaches the file event noise problem by ignoring file events which are unlikely to be related to the user's current task, inferring task from the currently focused application window. This applies two observations about desktop use: first, the focused application window generally captures the user's current interest within a task, and second, the set of recently focused application windows reasonably approximates the user's task [8]. Hence, FTF ignores all file operations which were not caused by the process of the currently focused window[1], and builds relationships between files which experience one or more access events by processes whose application window has gained focus within the recent past.

Due to noisy data, Connections employed a short (30 seconds), fixed length relation window. Because FTF's filtering substantially reduces the volume of file events considered, it can relax the constraints on the relation window. FTF starts a relation window when an application window gains focus and ends that relation window when the application window loses focus. This aligns 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 [12].

FTF maintains a log of relation windows for each window that was focused within the last *n* seconds (e.g., 300 seconds). When focus changes, the current relation window is ended and the relation graph is updated in two steps (depicted in Algorithm 1). First, FTF increases the link weights between each file experiencing an event within the relation window by the inverse of the count of unique files read or written during a relation window. This enhances the strength of the relationships between files during windows where few events occur. 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).

Confluence's FTF 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.

Algorithm 1 $processNewRelationWindow_{ftf}$: 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)
$reads_c \leftarrow getFilesRead(RW_c)$
$writes_c \leftarrow getFilesWritten(RW_c)$
{First, update between files in current window}
for all read file $r_i \in reads_c$ **do**
  for all read file $r_j \in reads_c$; $r_i \neq r_j$ **do**
    incrementGraph(r_i, r_j, $\frac{1}{|reads_c|}$)
    {$|reads_c|$ represents the number of unique files read}
  end for
end for{Repeat for writes}

{Now, update for each window in the log}
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 $r_j \in reads_c$ **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}

4.3 TaskRank

The *TaskRank* algorithm addresses the hidden activity problem. As described in Section 3, many applications employ configuration and state-maintenance files throughout their execution, transparently to the user. The frequency with which these *supernode* files are accessed increases both their connectedness in the relation graph and the weights of their individual links, causing them to falsely appear related to the user's task.

Manifested on the relation graph, supernodes feature a disproportionately high number of links and tend to bridge otherwise distinct tasks. In this respect, context-based search diverges from web search methods, such as PageRank, which infer authority or credibility for a page from a high number of incoming links. 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.

The TaskRank function calculates the exclusivity of the relationship between a given file and a given task-based file set. Equation 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 1 calculates the sum of the link weights from f to each file in $T \cap F$. The bottom half of calculates the total weight of all links from f.

$$TR(f,T) = \left(\frac{\sum_{f_i \in F \cap T} linkValue(f, f_i)}{\sum_{f_i \in F} linkValue(f, f_i)} \right)^2 \quad (1)$$

Conceptually, TaskRank represents the amount of a file's total link weight that is part of a given file set. A TaskRank value close

[1]Confluence also checks if the file event was caused by a "windowless" descendant process of the focused application; e.g., when a command-line shell — within a window — spawns a non-GUI process like LaTeX.

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

Table 1: Traced events

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.

GraphSearch is the method for finding contextually related files to a given file within the relation graph. To execute a GraphSearch on file f_A, we first retrieve the set of files F_A that are linked to f_A and assign each $f_i \in F_A$ a ranking-weight equal to the product of the link weight between f_A and f_i and the TaskRank value $TR(f_i, F_A)$. The files in F_A are then sorted by this rank, which indicates the relative strength of the contextual relationship.

Note that TaskRank does not exclude files from being part of multiple tasks; we see in practice that, in cases where a user's file is part of multiple tasks, it still yields 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.

5. EVALUATION

The goal of Confluence is to accurately identify contextual inter-file relationships that can be used by context-aware applications, e.g., context-enhanced search and browsing or context-based clustering. 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: a controlled study where users worked on a set of predetermined tasks and a longitudinal field study.

Data collection for each experiment used Confluence to trace the process, file system and user-interface events listed in Table 1. Confluence traces process creation and exit events to (1) maintain necessary mappings between a process's file descriptors and the files they represent and (2) maintain the process hierarchy required for FTF to identify ownership relationships between window and processes and subprocesses. From a performance standpoint, Confluence's tracing does not contribute a noticeable or significant performance penalty on modern personal computers.

Our evaluation compares the GraphSearch performance between Confluence and Connections. Relation graphs were generated using Connections's read/write algorithm (as described in [12]) and Confluence's FTF algorithm parameterized with relation window sizes of 5 and 10 minutes. To GraphSearch Connections's relation graph, we use its Basic-BFS algorithm (a weight and distance constrained variant of breadth-first search, described in [12]).

5.1 Experiment I

Our first experiment was a controlled user study where users were asked to complete one or both of two predetermined tasks involving computer use. In the first task, $task_1$, users authored a summary report for a fictitious conference. This task required users to create a document which described the conference, read a small number of ACM papers, and write small summaries of these papers on the summary document. In the second task, $task_2$, users created an online photo album with a small number of images (e.g., JPG, GIF) within a given topic (e.g., dinosaurs). First, users searched online for images, downloaded them, then uploaded them to an online web album (e.g., Flickr). The users completed the tasks in succession, using a laptop we provided.

Method	$task_1$	$task_2$	Combined
FTF (300 seconds)	0.542	0.682	0.574
FTF (600 seconds)	0.694	0.955	0.755
Connections	0.000	0.307	0.072

Table 2: Total recall in Experiment I

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 image file was considered part of the photo album task. Despite filtering file accesses to include only those from files under the user's home directory, the traces of Experiment I contained 2116 unique files, with relevant files making up only 4% of these. In total, we recruited *16* volunteers for our study; 13 volunteers accomplished $task_1$ and 13 accomplished $task_2$ (10 accomplished both), resulting in 53 relevant files for $task_1$ and 37 relevant files for $task_2$ (roughly, 3 files were used during each task, although some volunteers used more files).

5.1.1 Relation graph

For FTF (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. We consider the combined graph as a simulation of a single user performing a sequence of tasks, more closely resembling a single user's behavior over an extended period of time.

For each file f_i in a task T, we executed a GraphSearch on each of the methods' relation graphs to determine how many of the other relevant files within T that were present within its result pool. We define the metric *task recall* to be the percentage of files within T which are returned upon query f_i. Conceptually, *task recall* reflects the effectiveness of an algorithm at identifying file relationships with respect to task. In this experiment, we consider only the subset of the relevant files accessed by that user to be relevant for that search, while in the combined graph all relevant files are considered.

Table 2 lists the total recall values averaged across the $n + 1$ graphs for each method within each task (the recall-precision graph for FTF-600 is displayed in Figure 2, described later). FTF significantly outperforms Connections due to Confluence's filtering, which enables it to significantly expand its relation window. The increase in recall from a 300 second to a 600 second relation window FTF indicates that user tasks span long periods of time, further confirming our claim.

When asked to retrieve their files with Google Desktop (a traditional content-only search tool) most users were unable to generate keywords for at least one item within their task, although no user failed to identify keywords for at least one item in the task. This illustrates a need for Confluence's contextual retrieval; by remembering *any* item within their task, users would have had a strong chance of recovering any forgotten item by using keywords of a remembered item as a query.

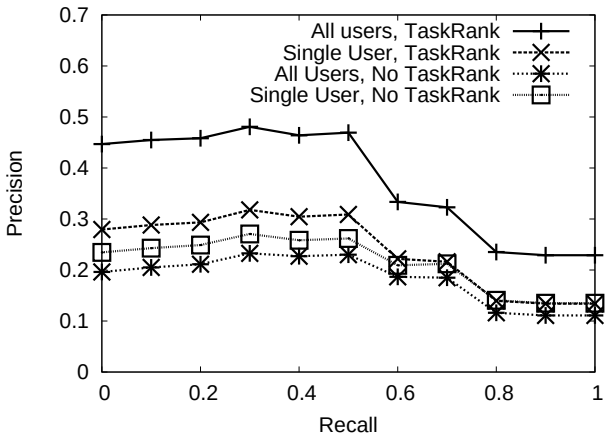

Figure 2: TaskRank on FTF-600 in Experiment I

5.1.2 TaskRank

Because TaskRank minimizes the impact of supernodes, 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.

Figure 2 illustrates the precision/recall 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 with more user activity. The fact that the worst performance was observed when considering all the users' data without TaskRank emphasizes the need for its noise reducing abilities.

5.2 Experiment II

Our second experiment evaluated the utility of Confluence's relation graph when generated 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. 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, after the tracing period completed, each user was asked to identify 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.

We used Confluence to perform a GraphSearch on 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[2]. 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.

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 task relationship (i.e., the files are used as part of the same task) to the seed file and 0 indicating no task 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. The remaining files were used as the ground-truth approximations for which we evaluated each algorithm's task-recall. As tasks can vary widely in number of files, we observed a variety of pool sizes.

	Result size					
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
Connections	0.11	0.13	0.16	0.17	0.17	0.17

Table 3: Task-recall by result size. Shaded cells indicate a statistically significant difference with Connections, derived from one-sided t-test ($df = 70$). Dark gray is significant with $P < 0.01$, light gray is $P < 0.05$.

	Result size					
Method	5	10	15	20	25	30
FTF (300)	0.46	0.31	0.27	0.25	0.23	0.21
FTF (600)	0.46	0.30	0.26	0.23	0.22	0.19
Connections	0.30	0.18	0.15	0.12	0.09	0.08

Table 4: Task-precision by result size.

5.2.1 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 across all users. The number of files in the approximate ground-truth averaged 13.66 per query.

Table 3 depicts each algorithm's task-recall at 6 result list sizes ranging from the first 5 to the first 30 results, averaged over the 36 seed files. Connections's performance is stronger in this experiment than in the previous one, confirming our expectation that it performs better when given more time to enforce the relationships generated by repeated behavior. However, Confluence still meets or exceeds Connections recall at all considered result sizes. At 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 used by Connections grows to nearly 30%. This improvement is significant based on a one-sided t-test using 70 degrees of freedom ($P < 0.05$ for all result sizes 15 or greater; $P < 0.01$ for all result sizes 25 and greater).

6. DISCUSSION AND CONCLUSIONS

Our experiments did not deploy Confluence as a complete search engine because we wanted to isolate the effectiveness of each individual method — hence, these reported numbers do not directly

[2]Additionally, these pools contained results from variations on the FTF algorithm which we do not describe in this work.

assess Confluence in natural document retrieval tasks. However, by situating our results within the original Connections evaluation, we draw more general inferences about the ability of Confluence to substantially improve end-to-end retrieval tasks. In the original Connections evaluation, users deployed the tracing and search software on their personal computers, using it whenever they needed to find a misplaced file. Hence, the results reported indicate how it performed in real, end-to-end retrieval tasks. In the Connections experiments, it was shown to improve recall by 34% to 74% without a reduction in precision when considering all returned results, but only increased recall from 18% to 34% when considering the first 30 results. These results substantiate the ability of context to greatly enhance personal document retrieval.

Our study directly addresses the core of file-based context building algorithms; namely, measuring the accuracy of our methods in identifying file relationships. Though our users did not use Confluence as a search system, we generalize that our significant improvements to the context-building core that we would translate into substantial improvement in typical, end-to-end document retrieval tasks.

From our experiments and experience with the system, we draw the following points:

- In cases where users remember some but not all files from a task, Confluence would be an effective retrieval tool. This case presented in our 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.

7. REFERENCES

[1] 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 Press.

[2] A. N. Dragunov, T. G. Dietterich, K. Johnsrude, M. McLaughlin, L. Li, and J. L. Herlocker. Tasktracer: a desktop environment to support multi-tasking knowledge workers. In *IUI '05*, pages 75–82, New York, NY, USA, 2005. ACM Press.

[3] S. Dumais, E. Cutrell, J. Cadiz, G. Jancke, R. Sarin, and D. C. Robbins. Stuff i've seen: a system for personal information retrieval and re-use. In *SIGIR '03*, pages 72–79, New York, NY, USA, 2003. ACM Press.

[4] 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.

[5] V. Kaptelinin. UMEA: translating interaction histories into project contexts. In *CHI '03*, pages 353–360, New York, NY, USA, 2003. ACM Press.

[6] D. Karger, K. Bakshi, D. Huynh, D. Quan, and V. Sinha. Haystack: A general purpose information management tool for end users of semistructured data. In *CIDR*, pages 13–26, 2005.

[7] T. 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.

[8] N. Oliver, G. Smith, C. Thakkar, and A. C. Surendran. SWISH: semantic analysis of window titles and switching history. In *IUI '06*, pages 194–201, New York, NY, USA, 2006. ACM Press.

[9] T. Rattenbury and J. Canny. CAAD: an automatic task support system. In *CHI '07*, pages 687–696, New York, NY, USA, 2007. ACM Press.

[10] P. Ravasio, S. G. Schär, and H. Krueger. In pursuit of desktop evolution: User problems and practices with modern desktop systems. *ACM Trans. Comput.-Hum. Interact.*, 11(2):156–180, 2004.

[11] J. Shen, L. Li, T. G. Dietterich, and J. L. Herlocker. A hybrid learning system for recognizing user tasks from desktop activities and email messages. In *IUI '06*, pages 86–92, New York, NY, USA, 2006. ACM Press.

[12] 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 Press.