A Case Study of Paired Interleaving for Evaluating Code Search Techniques

Kostadin Damevski
Virginia State University
Petersburg, VA 23806
kdamevski@vsu.edu

David Shepherd
ABB, Inc.
Raleigh, NC 27606
david.shepherd@us.abb.com

Lori Pollock
University of Delaware
Newark, DE 19350
pollock@udel.edu

***Abstract*—Source code search tools are designed to help developers locate code relevant to their task. The effectiveness of a search technique often depends on properties of user queries, the code being searched, and the specific task at hand. Thus, new code search techniques should ideally be evaluated in realistic situations that closely reflect the complexity, purpose of use, and context encountered during actual search sessions. This paper explores what can be learned from using an online paired interleaving approach, originally used for evaluating internet search engines, to comparatively observe and assess the effectiveness of code search tools in the field. We present a case study in which we implemented online paired interleaving for code search, deployed the tool in an IDE for developers at multiple companies, and analyzed results from over 300 user queries during their daily software maintenance tasks. We leveraged the results to direct further improvement of a search technique, redeployed the tool and analyzed results from over 600 queries to validate that an improvement in search was achieved in the field. We also report on the characteristics of user queries collected during the study, which are significantly different than queries currently used in evaluations.**

***Keywords*-code search, feature location, online paired interleaving, field studies**

I. Introduction

Locating the code related to a specific feature or concern is often a software developer's first step in performing a software maintenance task [1]. Ko et al. [1] conducted a study that showed that out of 48 instances of developers working on tasks, 40 began with a textual search. To aid software engineers in this activity, researchers have developed a number of Code Search Techniques (CSTs), including static, dynamic, and hybrid approaches that identify relevant code, which is often scattered across a large, complex software system [2], [3], [4], [5]. As evidenced by many recent publications [2], [6], [7], CST innovation is often directed by analyzing the results from studies of current approaches on gold sets, which contain search queries and their associated results as a set of relevant code units. Similarly, improvements on underlying text analyses for code search techniques, such as identifier splitting, are also driven by gold set evaluations. In both cases, effectiveness is measured in terms of precision, recall and F-measure using the gold set of code units as an oracle. The validity of these comparative studies relies heavily on the quality of these gold sets, as both missing or incorrect code units can easily lead to misguided conclusions.

Unfortunately, constructing high-quality gold sets presents many challenges. One common method for creating gold sets is to extract queries from bug reports (or collect queries from the field) and then, for each query, gather human judgements on which code units are relevant. This approach raises several concerns: (1) it may not capture real developer intent as the gold sets are typically generated outside the search context; (2) the gold sets are often produced by researchers, not actual developers, making them one step farther from the actual use case; (3) researchers have observed that human judges vary in their opinion of what comprises the relevant code to a particular feature and thus there is subjectivity [8]; (4) researchers have also found that the relevant code to a particular feature might be dependent on the maintenance task to be performed and the developer himself, implying the gold set is context dependent [8].

These concerns warrant complementing gold set based evaluation of CSTs in the lab with field evaluation of CSTs during daily maintenance tasks in order to both more accurately assess their effectiveness and to help better understand their defficiencies. In addition to gaining access to queries submitted by developers in the field, more importantly, information about their actual behavior in response to queries can be recorded to better estimate their intent. This is especially useful feedback because the effectiveness of a search technique often depends on properties of the user queries and the code being searched, and search results to a single query can affect the user's future search behavior and subsequent queries. One way to determine the relative performance of code search techniques is through implicitly-gathered user behavior during regular maintenance tasks and by leveraging a paired interleaving technique first proposed to evaluate Internet search engines [9]. Others have advocated a comparative assessment approach for other software engineering tasks, including measuring software quality [10].

This paper explores what can be learned by using the paired interleaving approach to comparatively observe and assess the effectiveness of CSTs in the field, during normal software maintenance. Paired interleaving's comparative study design is similar to sensory analysis, where the tastes of two products are compared by asking the test subjects to express a preference for a particular product, instead of rating each of the products on an absolute (i.e., Likert) scale. Two CSTs are implemented within the same CST user interface, and results from the two CSTs are presented to the user interleaved alternately unbeknownst to the user, as they make queries. Users' preferences for results from different queries are collected in the form of user clicks on specific results from the interleaved result list and analyzed. Thus, implicit feedback is gathered from actual developers in the context of their normal software maintenance process, and large deployment of the tool can enable significant data collection.

We present a case study in which we implemented paired interleaving for code search, deployed the tool in an IDE for developers at multiple companies and analyzed results from over 300 user queries (18 per development day) during their daily software maintenance tasks. We leveraged the results to direct further improvement of a search technique, redeployed the tool and analyzed results from over 600 queries (25 per development day) to validate that indeed an improvement in search was achieved in the field. In this paper, we report on our experiences and lessons learned with using such a tool to compare and improve code search technology. The main contributions of this paper are:

1) a novel approach to gaining comparative, implicit user feedback on CSTs during regular software maintenance tasks;
2) description of the challenges and benefits in instantiating the paired interleaving approach;
3) results from a case study of using paired interleaving to evaluate two CSTs, including CST comparison results, effects of design choices in implementing paired interleaving for CST comparison, observations from user queries in the field, and
4) case study experiences in how this approach can be used to identify improvements in code search techniques.

We describe the paired interleaving evaluation technique in Section II. The results of the case study of paired interleaving are discussed in Section III, while Section V describes related work.

II. Paired Interleaving for CST Comparison

A. Overview

The key challenge in basing evaluation on usage data instead of expert judgement is properly interpreting the gathered data through carefully relating the observable statistics to the quality of the techniques under evaluation [11]. When applied to evaluate Internet search engines, paired interleaving achieves an online comparison of different result sets associated with different search engine techniques. The two result sets, each from different search engine technologies, are merged into a single interleaved set, which is presented to the user such that the observed user behavior, in the form of clicks, is indicative of a preference for a particular search engine.

We believe that paired interleaving can also be applied to evaluate the relative effectiveness of two CSTs, or CSTs with different text analysis or preprocessing. The evaluation is conducted by recording implicit feedback (user click data) during daily software developer interaction with a code search tool that embeds and interleaves the responses of different complete approaches to code search. The relative preference for one approach over another is indicated by a preference for the results of one CST over another, in the span of a set of queries by a number of developers.

A set of properties, initially discussed by Joachims [11], are required for the paired interleaving process to be fair and robust to bias. First, while the CST's backend is augmented to produce results of two separate CSTs for each developer query, the tool's frontend and the the developer's search experience must remain unaltered. Second, the two sets of CST results must be interleaved fairly and presented to the developer as a unified list, with no indication of the origin of each result. Thus configured, the evaluation experiment can determine developer preference for a specific CST by counting developer clicks on its results and using rankings within the original result lists from the different code search techniques.

B. Balanced Interleaving

Consider the process of evaluating, via paired interleaving, CST_A and CST_B, which produce result sets $A = \{a_1, a_2, ..., a_n\}$ and $B = \{b_1, b_2, ..., b_n\}$ for a particular developer query q. A reasonable assumption is that each of the results sets are ranked in descending order of relevance to q. The evaluation is performed using a set of queries, $q_1, q_2, ..., q_m$, where m is sufficiently large to establish a statistically significant preference for a particular CST. Interleaving the two result sets A and B produces an interleaved set I, also of size n, instead of twice that, in order to ensure that the developer's experience remains unaltered.

A common approach to performing the interleaving, called Balanced Interleaving, is presented as pseudo code in Figure 1 [11]. This algorithm randomly determines whether the interleaving begins with a result from CST_A or CST_B, and then proceeds by inserting a non-duplicate result from each CST. Table I shows two example original results sets, A and B, from running CST_A and CST_B, respectively for a given query, and the displayed Balanced Interleaving sets depending on whether the first element comes from list A or list B. While the results are interleaved from both lists, elements that appear in both A and B are only listed once, and thus, often a given element in the displayed list appears in both original lists.

To determine the user preferences for a given search technique, the set of clicks for each query is analyzed

Rank	A	B	Interleaved A first	Interleaved B first
1	a	b	a	b
2	b	e	b	a
3	c	a	e	e
4	d	f	c	c
5	g	g	d	f
6	h	h	f	d
7	i	j	g	g
.	.	.	.	.
.	.	.	.	.

Table I
EXAMPLE INTERLEAVED RESULT SETS [9]

```
Input: search result lists A and B from $CST_A$, $CST_B$
1: $I$ := {}; // initially empty Interleaved list
2: $k_A$ := 0; $k_B$ := 0; // start at first result from each list
3: n := A.length(); // Interleaved same size as orig lists
4: AFirst := RandomBit(); // coin toss for first result
5: while $k_A + k_B < n$ do
6:     if $k_A < k_B$ or ($k_A == k_B$ and AFirst) then
7:         if not I.contains(A[$k_A$]) then
8:             I.insert(A[$k_A$]);
9:         end if
10:        $k_A$ := $k_A$ + 1;
11:    else
12:        if not I.contains(B[$k_B$]) then
13:            I.insert(B[$k_B$]);
14:        end if
15:        $k_B$ := $k_B$ + 1;
16:    end if
17: end while
Output: interleaved list I
```

Figure 1. Pseude-code of the Balanced Interleaving.

and counted as a win for one of the search techniques or a tie. The per-click scores are aggregated into per-query preferences, which, in turn, are combined into a single metric that expresses the entire Balanced interleaving experiment's result.

In our implementation of Balanced Interleaving, we score each click on the interleaved set I as either: the result set A (i.e. CST_A) wins, the result set B (i.e. CST_B) wins, or neither wins (tie). To determine wins for each clicked result r in the interleaved result set I, we locate r in each of the original result sets, A and B, and compare the ranks in the original lists. CST_A is awarded a win if r is ranked closer to the top of list A than in list B, and vice versa. If r is ranked at the same location in both lists A and B, then the click is scored as a tie. If a result r does not appear on a given original result list, it is considered to be a win for the other search technique. For example in Table I, a click on a in the interleaved list in column $Interleaved\ A\ first$ would count as a win for CST_A, while a click on b or e would be wins for CST_B, and clicking on g would be a tie. These per-click results are aggregated into wins, losses or ties at the query level, encapsulating the developer's interaction with an interleaved result set. This is performed by using the wins for a particular CST as the deciding factor, i.e. more per-result wins for CST_A than CST_B would result in a per-query win for CST_A. The scoring mechanism could easily be replaced by a different one, such as in [9].

Similar to the balanced interleaving approach for Internet search engine evaluation, we use a single statistic to quantify the degree of preference for a particular CST and summarize the outcome for an entire interleaving experiment [9]. The following equation defines the statistic Δ_{AB}, where $wins(A)$ and $wins(B)$ are the number of times clicks show preference for CST_A and CST_B, respectively, based on the methodology just described. A positive value for Δ_{AB} indicates a preference for CST_A, a negative value a preference for CST_B, while the magnitude of this metric is indicative of the strength of the preference.

$$\Delta_{AB} = \frac{wins(A) + \frac{1}{2}ties(A,B)}{wins(A) + wins(B) + ties(A,B)} - 0.5 \qquad (1)$$

In addition to producing a measurement for the preference for a particular CST (i.e. Δ_{AB} in equation 1), it is useful to have a confidence bound for this measurement. This confidence bound can be useful in determining whether the results of a Balanced Interleaving experiment are statistically significant as well as determine whether gathering more samples could be beneficial. To determine the confidence bound, without making assumptions about the distribution of the data, bootstrapping estimators can be used. A more detailed description on applying bootstrapping to Internet search results can be found in Chapelle et al. [9].

III. CASE STUDY

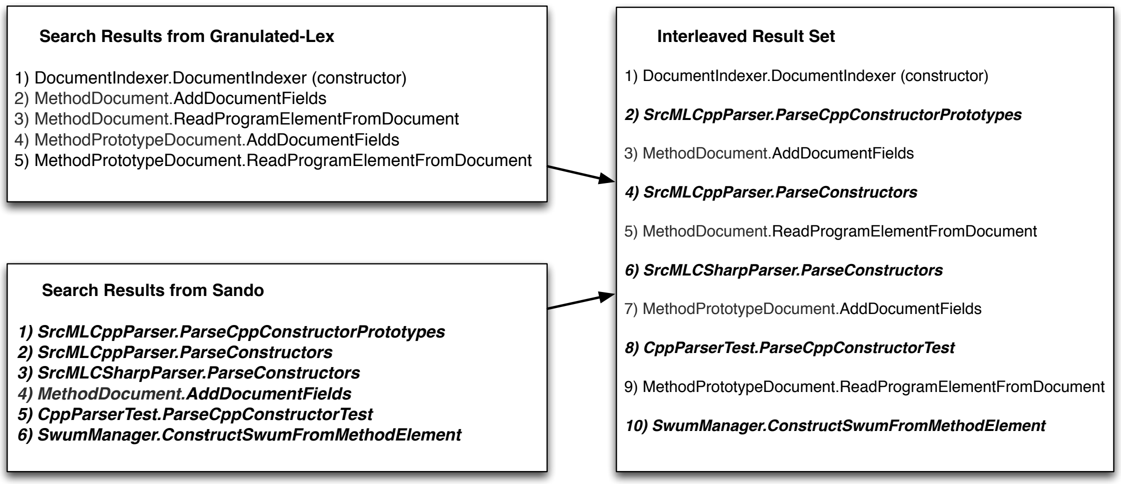
We designed and conducted a case study aimed at investigating three main research questions:

1) How well does online paired interleaving work in practice as an approach to evaluating and identifying improvements of code search techniques?
2) In applying paired interleaving for code search evaluation and improvement, how should one account for the difference in corpus size versus Internet search and the variations in code search techniques under comparison?
3) What can be learned about queries recorded in the field as users submit them during their daily software maintenance work and how do those observations differ from characteristics of gold sets?

A. Paired-interleaving Implementation

We exploited the extensibility of the Sando code search tool [7] to create a tool that implements paired interleaving for CST evaluation. Sando is an open-source code search tool [1] based on Information Retrieval (IR) search technology.

[1]http://sando.codeplex.com


Search Results from Granulated-Lex

1) DocumentIndexer.DocumentIndexer (constructor)
2) MethodDocument.AddDocumentFields
3) MethodDocument.ReadProgramElementFromDocument
4) MethodPrototypeDocument.AddDocumentFields
5) MethodPrototypeDocument.ReadProgramElementFromDocument

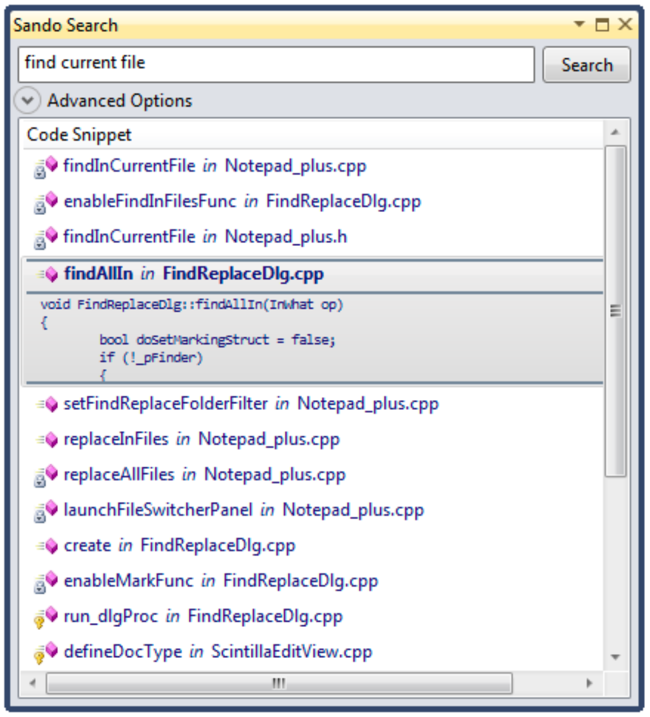
Search Results from Sando

1) SrcMLCppParser.ParseCppConstructorPrototypes
2) SrcMLCppParser.ParseConstructors
3) SrcMLCSharpParser.ParseConstructors
4) MethodDocument.AddDocumentFields
5) CppParserTest.ParseCppConstructorTest
6) SwumManager.ConstructSwumFromMethodElement

Interleaved Result Set

1) DocumentIndexer.DocumentIndexer (constructor)
2) SrcMLCppParser.ParseCppConstructorPrototypes
3) MethodDocument.AddDocumentFields
4) SrcMLCppParser.ParseConstructors
5) MethodDocument.ReadProgramElementFromDocument
6) SrcMLCSharpParser.ParseConstructors
7) MethodPrototypeDocument.AddDocumentFields
8) CppParserTest.ParseCppConstructorTest
9) MethodPrototypeDocument.ReadProgramElementFromDocument
10) SwumManager.ConstructSwumFromMethodElement

Figure 2. Example paired interleaving of search results for the user query 'const'.



findInCurrentFile *in* Notepad_plus.cpp
enableFindInFilesFunc *in* FindReplaceDlg.cpp
findInCurrentFile *in* Notepad_plus.h
findAllIn *in* **FindReplaceDlg.cpp**

```
void FindReplaceDlg::findAllIn(InWhat op)
{
        bool doSetMarkingStruct = false;
        if (!_pFinder)
        {
```

setFindReplaceFolderFilter *in* Notepad_plus.cpp
replaceInFiles *in* Notepad_plus.cpp
replaceAllFiles *in* Notepad_plus.cpp
launchFileSwitcherPanel *in* Notepad_plus.cpp
create *in* FindReplaceDlg.cpp
enableMarkFunc *in* FindReplaceDlg.cpp
run_dlgProc *in* FindReplaceDlg.cpp
defineDocType *in* ScintillaEditView.cpp

Figure 3. Screenshot of Sando showing search results.

Its modular structure allowed us to create an implementation that reacts to a given query by retrieving result sets from both Sando's IR-based search technique as well as Visual Studio's state-of-the-practice technique (the next subsection discusses why these CSTs were chosen), while providing the user with a common interface and without exposing the mapping between each individual result and its underlying search technique. Users interact with this CST evaluation tool through Sando's usual user interface, where they are presented with a list of results matching their query, internally compiled from both CSTs under evaluation according to the algorithms in Section II. Figure 3 shows a screenshot of Sando with query and returned results listed.

To illustrate the presentation of search results, Figure 2 shows two search tools' first 5-6 returned results, at the method and class level, for the user query `const`. The results are interleaved using the Balanced Interleaving algorithm to produce the search result set displayed to the user, who is kept unaware that two separate tools' results were combined to produce the result set. In order to keep the display of results similar between the two search tools, both the result sets are normalized to the same granularity level. Individual results that are retrieved by both search tools (`MethodDocument.AddDocumentFields` in Figure 2) are not duplicated in the interleaved result set.

The user preference for a given search technique is measured using the formulas given in Section II for computing wins, losses, and ties based on the user-clicked results. Sando's user interface differentiates a single-click on a result from a double-click; a single-click displays a 5-line relevant snippet of code to the user, while a double-click opens the entire source file in the Visual Studio editor placing the cursor on the matching line. In our CST evaluation tool, we monitor only the participants' double-clicks, which are more strongly indicative of user interest in a specific result than single-clicks. Based on our observation of developers, the availability of the code snippet in our tool filters out spurious double-clicks on the result set and focuses the developer double-clicks only to relatively promising result items.

B. Code Search Techniques under Comparison

For this case study, we chose search techniques for initial comparison to satisfy the following criteria: publicly available, representing state of the practice search techniques, and implemented in C# within Visual Studio, to easily incorporate within our Sando setup. Because we also wanted to investigate the ease in implementing the paired interleaving approach with CSTs, we chose techniques that were not a variation of the same overall approach (e.g., two IR-based approaches). The first search technique we use in this case study is Sando's own searching mechanism. SANDO operates by parsing the source code of a particular software project into a set of program elements (methods, classes, fields, etc.), creating bag-of-words documents by splitting identifiers in the source code belonging to each of these program elements. Program elements are then placed in an index and subsequently retrieved via a user query. SANDO uses TF*IDF scoring. SANDO weighs program element names (e.g., class and method names) higher than their bodies, as the names are often more indicative of the element's purpose. The tool is implemented as a Visual Studio extension and uses Lucene [2] for its index.

For a comparison search technique, we chose to use the Visual Studio FIND IN FILES tool, which likely represents the most commonly used search technique for Visual Studio. FIND IN FILES uses a simple lexical matching between the query and the code. By default, it reports all occurrences of matches, at the line level, while SANDO reports results at the method, field, comment and class levels.

To ensure that all results look similar in form to the user, the interleaved result list needs to be reported at the same granularity for all results. We report the retrieved results at the larger granularity level of SANDO instead of FIND IN FILES's line level because the one technique, SANDO, does not provide information at the finer level. If FIND IN FILES finds multiple matches within the same method, then at this higher granularity level, the result set would contain duplicate entries. For example, the result list might look like $M_1, M_1, M_1, M_3, M_3, M_5$. The appearance of duplicate results in the result list could bias the evaluation as the user would be able to recognize the repeated results as coming from a specific code search technique. It may also lead to user frustration in having to consider a larger set of results, leading to quicker than otherwise abandonment of that specific result set to reformulate the query.

To address this granularity difference, we map the results retrieved from FIND IN FILES to their containing program elements. If the match occurs within a method, the match is reported as the containing method. If the match occurs within a comment at the class level, the result is reported at the class level. This mapping helps to disguise the fact that there are two underlying CSTs being used to produce the result list, making the evaluation less biased. We call this CST the GRANULATED-LEX technique. This also effectively reduces the number of reported results for the FIND IN FILES technique. GRANULATED-LEX tends to produce a more manageable number of results than FIND IN FILES, but still provides no ranking and likely produces no results to multi-word queries. Figure 4 depicts the design of the CST evaluation tool instantiated to compare our targeted search mechanisms.

[2]http://blogs.apache.org/lucenenet/

C. Participants and their Software

A total of 6 professional developers in 3 different companies were recruited to participate in the study. The developers used the provided CST evaluation tool as a Visual Studio plugin to maintain some of their company's proprietary software. All of the developers used C# and had at least 3 years of experience with the language, and at least 5 years of programming experience. The developers had previously been using Visual Studio for their development and continued to do so during the study. They had no familiarity with Sando in particular or other CSTs in general.

Some of the developers were working with codes that contained identifiers in a Slavic language, thus some queries and identifiers were not in English. This provided opportunity to investigate the differences in performance of the CSTs for different natural languages.

D. Procedure

At the outset of the study, the participants were given a simple set of instructions to download and install the CST evaluation tool, which was distributed as a Visual Studio extension. The instructions to the participants consisted of an explanation of when and how they might use what looked like a search tool to them:

"This tool is best used in software maintenance tasks, where you are looking for the initial code location to fix a bug or to implement a new feature. For our study, you don't have to do anything special - just use the search tool for parts of your daily work where you think it will be most useful."

Defining the usage scenario for the CST evaluation tool more narrowly to our participants, such as a firm request to only use the tool at the beginning of a code maintenance task, may have biased the normal usage results and caused results not applicable to broader usage.

Data on usage of the CST evaluation tool was collected automatically. In particular, the count of user double-clicks on each CST's result set was logged, and this log was periodically and automatically uploaded to our server. The logs were organized by the participant's machine name, which was correlated to each participant at the end of the first study period. Queries for which there were no double clicks on the result listing were ignored. For more detailed data analysis of the results for each query, we also logged the query string, the number of results returned by SANDO and GRANULATED-LEX for that query string, and per-click information including the kind of program element that was

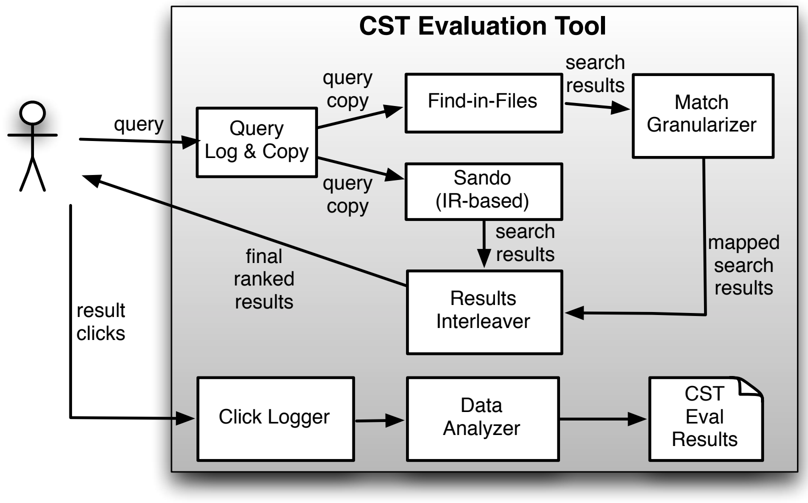


Figure 4. Implementation of CST evaluation tool.

clicked and whether the clicked element contained an exact or partial match of the query term.

To gain insight into our first research question, "how well online paired interleaving works in practice as an approach to evaluating and identifying improvements of CSTs", we first used paired interleaving to compare SANDO and GRANULATED-LEX. We calculated the number of wins for each CST and the ties. The confidence interval calculation was based on the bootstrap percentile statistical method, as in Internet search [9]. Bootstrap percentile requires a large number of entries n before it can accurately estimate the confidence interval with p confidence. We used the rough estimate of $n(1-p) \geq 5$, which corresponds to 100 queries at 95% confidence, in order to constrain the number of entries needed to calculate the confidence interval. To explore how the search techniques perform comparatively for queries in English and Slavic, we gathered our results for each language separately, and report them both separately and in aggregate.

We tracked the time-based results of the evaluation (as a whole, and per language) each day, by the calculated Δ_{AB} metric and its error range for all of the queries cumulatively up to that day of the study. This assessed whether we had a conclusive result at the end of each day. We stopped collecting data when we obtained a clear indication of one technique winning. The study comparing GRANULATED-LEX and SANDO was concluded after 325 queries as the entire interval was in the GRANULATED-LEX range (the entire error bar was in the negative part) on day 7, and the number of received queries was significant and passed a 100 query threshold, below which we deemed the confidence interval calculated by the bootstrap percentile method as not trustworthy.

We then explored the use of paired interleaving as a tool for evolving a CST by first analyzing the results of the GRANULATED-LEX versus SANDO study, with the intent of learning how we might improve the search technique behind SANDO. By comparing the results of the two CSTs in the field through the user's clicking behavior, we hypothesized that SANDO could perform better for the users in several ways. To test our hypothesis in the field, we created a new version of SANDO, which we call ENHANCED-SANDO that includes the unsplit version of identifiers in the indexing along with the split versions, and adds additional weight to documents that match the query exactly. A search for `openFile` will match method `openFile` more strongly than `openTheFile`. Participants were asked to install a new version of the CST evaluation tool, and we started collecting data from the field again. In this second study, we continued collecting queries for 14 days, collecting over 600 queries in total, but ending with no conclusive winner or loser (more details in the Results section). We considered letting the second study go longer, but it was not converging on a winner. We also wanted to keep the number of queries in the same order of magnitude to make a decision based on similar size data sets.

IV. CASE STUDY RESULTS

We present several categories of results. We start by presenting the paired interleaving results from our two case studies, first SANDO versus GRANULATED-LEX, and then ENHANCED-SANDO versus GRANULATED-LEX. Next, we present these same results broken down by language.

Comparative Studies: Table II presents the final results from both comparative studies using the paired interleaving technique. In the end, the participants submitted 325 queries in the first study and 637 queries in the second study. The first study, of SANDO versus GRANULATED-LEX concluded

Study	# Queries	# SANDO/ENHANCED-SANDO Wins	# GRANULATED-LEX Wins	# Ties	Δ_{AB}
SANDO V. GRANULATED-LEX	325	106	143	76	**-0.057**
English queries only	152	54	65	33	**-0.036**
Slavic queries only	173	52	78	43	**-0.075**
ENHANCED-SANDO V. GRANULATED-LEX	637	239	222	176	0.013
English queries only	317	127	101	89	**0.041**
Slavic queries only	320	112	121	87	-0.014

Table II
FINAL RESULTS OF COMPARATIVE EVALUATIONS THROUGH PAIRED INTERLEAVING. BOLDED Δ_{AB} METRICS INDICATE THAT THE RESULT IS CONCLUSIVE AT 95% CONFIDENCE.

with GRANULATED-LEX being chosen as the superior retrieval algorithm by the participants.

The second study, of ENHANCED-SANDO versus GRANULATED-LEX concluded in a tie, and while ENHANCED-SANDO had a slight edge, it was not significant enough to declare it a winner with 95% confidence. The confidence interval was not entirely either positive or negative in this study; however, a positive value for Δ_{AB} indicated a preference for ENHANCED-SANDO despite not having enough data to conclude a clear win for ENHANCED-SANDO after 14 days of study and 637 queries.

These results indicate that the enhancements to SANDO (i.e., including the unsplit version of identifiers in the indexing, and adding additional weight to documents that match the query exactly) improved its performance in the field, with 99% confidence. In addition, we found the data analysis from this comparative assessment in the field to be quite helpful in revealing deficiencies that one CST may suffer from in the field.

Comparative Studies by Language: In the first study, of the 325 total queries, 152 were in English and 173 were in a Slavic language. In the second study, of the 637 queries, 317 were in English and 320 were in Slavic. As shown in Table II, at 95% condifence level, GRANULATED-LEX won on both English and Slavic queries in the first study, while ENHANCED-SANDO overcame GRANULATED-LEX on English queries in the second study. The GRANULATED-LEX versus ENHANCED-SANDO results were inconclusive when only Slavic queries were considered in the second study. Both ENHANCED-SANDO and SANDO performed significantly better (with over 99% confidence) on English queries than on Slavic queries. Conversely, in both studies, GRANULATED-LEX performed better on Slavic queries than on English queries. SANDO and ENHANCED-SANDO doing worse on non-English queries than English queries is expected, as SANDO's stemming targets the English language. Its word splitting is currently on camel case only, so it should not be affected by the query language.

Since there happened to be similar percentages of queries in each language, the overall results should not have been swayed by one query language.

V. BROADER LESSONS LEARNED AND DISCUSSION

In this section, we provide answers to the research questions that initiated the case study. We present a set of lessons learned from applying paired interleaving to code search evaluations, and finally a set of observations from reviewing the collected user queries.

A. *Lessons Learned from Applying Paired Interleaving*

The Paired Interleaving evaluation method was originally designed to evaluate Internet-scope search engines. In that context, there were two underlying assumptions affecting the evaluation design, namely:

- The **scope** of each search engine was an extremely large corpora (i.e., the contents of the Internet at that time).
- The **variation** between each search engine and user interface was relatively low.

When applying this evaluation technique to CSTs, these assumptions do not hold and thus we had to adjust the approach for our targeted software engineering problem.

Adjusting For Scope: When searching an Internet-scope corpus, most user queries return at least some results, and thus during a paired interleaving evaluation of search engines, any query that returns no results for *either* of the competing approaches is ignored. However, when searching a much smaller corpus (i.e., a single project), user queries often result in no results. To investigate this issue, we first implemented the two search techniques under evaluation within the paired interleaving evaluation tool without handling this situation explicitly. We observed that the situation of one technique returning no results for a given query occurred somewhat frequently during our initial testing. For instance, in the study of GRANULATED-LEX and ENHANCED-SANDO, 13.7% of the searches returned no results, (10.8% cases where ENHANCED-SANDO returned no results and 2.9% where GRANULATED-LEX returned no results).

Because one of the competing CSTs failed for a significant percentage of queries, we chose to handle this case explicitly. One approach is to, as in Internet search evaluations, ignore the results of one CST if the other retrieves no results, since this happens very infrequently. However, when evaluating CSTs, this will lead to bias against CSTs that return results even when others cannot. The alternative is to proceed as normal when one CST fails, effectively only returning results from the other CST. In this case, any click is indicative of a win for the CST that returned results and a loss for the CST that failed. We believe the second approach is the most fair because (1) if the returned results are poor, they will usually not be clicked on anyway and (2) if these results are not considered, then a very conservative CST (i.e., only returning results when its confidence is high) will always appear to outperform more optimistic CSTs.

While these failure cases were often ignored during internet-scale search studies, we have found that these cases are the most interesting cases during CST evaluations. The comparative approach identifies cases where one CSTs fails yet another does not, which are cases that have at least one known solution (i.e., the successful CST), and thus can always lead to improvement in the failing CST. As a comparison, when an CST fails during a gold set evaluation, there is not necessarily a known, feasible solution. However, a paired interleaving evaluation naturally leads to actionable findings.

Accounting for Variations: When searching the Internet using different search engines, the user is presented with a highly standardized experience. Almost all Internet search engines return the same type of results (e.g., a ranked list of links with summaries) in about the same amount of time (e.g., 1-5 seconds) using the same input (e.g., a query string). Unfortunately, CST tools are much less standardized, and their variations can affect the evaluation.

A detailed, manual analysis of our data showed that when one technique was very slow in returning results, users often quit their search and started a new search with a different query. Thus, slow performance by an CST can affect normal user behavior which is what the paired interleaving evaluation is trying to capture. *The altered user behavior, caused by slow performance of one CST, may bias against faster CSTs, and should be considered during data analysis.*

Previously, we described the granularity mismatch issue and how we dealt with it for the CSTs that we used in this case study (See Section III-B). In our case study, GRANULATED-LEX considerably reduced the number of results reported from FIND IN FILES by changing from line level to method and class level granularity. This change in granularity dramatically increased the effectiveness of the FIND IN FILES approach by eliminating much of the noise (i.e., multiple matches in a single method or class) from the results. *Granularity of how a technique reports its results can significantly improve users' preference for that technique.*

These observations, through comparative study, help to learn how different search techniques will perform in the field under daily use.

B. Characteristics of User Queries

The main focus of our case study was on the paired interleaving approach to evaluating CSTs. However, the developers also consented to our collection of the query strings they executed as they performed their daily tasks. By reviewing these query strings, we were able to learn more about the strategies and concrete use cases of the users. Specifically, we learned that:

- Developers use queries that are often ***a single word***.
- Users often search for ***known entities***.
- There can be significant differences in ***gold set queries vs. user queries observed during daily tasks***.

Query Length: We noticed that most queries are a single word[3]. In our data, the median number of words in a query over all the 637 queries was 1.0, with the mean being 1.08 words per query, with a standard deviation of .58.

While most queries consisted of only a single word, several queries were not even a single word long. Participants periodically issued the partial word queries (21 of 637 queries, or 3% of the queries were partial words), even though the CST evaluation tool's interface clearly indicates that only complete words should be provided as queries. For example, a user search for the Slavic word for days (`denovi`) used the query `den` instead of `denovi` (which she later queried). All of these partial queries returned no Sando results and each of the clicked program element results did not contain a whole word match with the query string. Partial word queries worked just as well for *Find in Files* as full word queries did. Because of this discrepency, GRANULATED-LEX won almost all partial word queries (all 21 cases) including the `denovi` case mentioned above.

Searching for Known Entities: CSTs are often designed to assist users in mapping from a high-level, domain concept to program elements [2], often as a precursor to fixing a bug in unfamiliar code. However, users' queries suggest that they also use CSTs to lookup known entities, using the search tool as a faster alternative to clicking through the file system to a known file or element. The queries show that they are often looking for specific identifiers, as indicated by camel cased queries, (e.g., one user queried for `PrepareControlForExport`), or as indicated by other special characters attached to the term (e.g., one user queried `mail.To.Add(email)`). There were a large number of queries, 186 of the 637 (30%), that are either in camel case and/or contain special characters to help indicate what type of element the user is searching for.

Similarly, we observed that many queries exactly match the search results that developers click on, indicating that developers were using the search tool to lookup the definition or usage of a known program element instead

[3]Here, we consider words to be space delimited strings, instead of their natural language connotation, such that *find* and *openFile* are both words.

of searching for a concept. For instance, after a search for `ExportToExcel`, a user selected a program element that exactly matched this string, likely the method `ExportToExcel`. In this case, the extra processing that SANDO and other approaches offer, including word stemming, identifier splitting, and even term weighing were simply not necessary, and thus a lexical search was sufficient.

To determine how often queries matched exact results, we considered the recorded similarity measurement between the clicked search results and the query and found that a large percentage of queries (522 of 637 queries, or 82%) had an exact match with the name or within the body of the program element. This suggests that an effective CST must handle this type of query well.

User Queries vs. Gold Set Queries: For comparison sake, we examined the differences between the user-created queries in this study and the researcher-created queries used in other studies. In particular, we compared against the queries from TraceLab's experimental data [6], which covers five separate open source projects, because they represent the most well-designed available gold sets. The difference was striking, as the queries used in TraceLab had around seven words on average [4], while we see an average of slightly over 1 word per query in our user-collected data. Studies of Internet search are closer to our data set with the average query consisting of between 1-3 words [12]. Starke's recent developer study observed that on maintenance tasks, developers tended to prefer a rapid trial-and-error approach to code search, ofter relying on query reformulation rather than a long, carefully constructed query [13]. Additionally, because the TraceLab dataset's queries are extracted automatically from a bug-tracking system (i.e., by taking the title text of a given bug), these queries are qualitatively different from our queries. For example, one of their queries is `[Keybindings] Wrong behaviour when triggering Code Assist in "Rename Field"-Dialog` whereas three examples of collected user-created queries are `upload`, `2009`, and `excelPath`. Similarly, the TraceLab data set has few known entity-style queries, whereas in our data these queries (e.g., `GridViewDataTextColumn`) made up a large percentage of the overall set.

C. Threats to Validity

As in most case studies, this case study was undertaken "...to investigate contemporary phenomena in [its] natural context" [14]. Because of this decision, there were naturally limits on our ability to collect statistically representative samples and to control the environment, as would be possible in a laboratory setting. For instance, a larger sample of developers may improve generalization; however, our developers were selected from several different companies. Since developers were not indicating what maintenance task they were performing when they performed their code searches, the variety of maintenance contexts during the study is not known. We focused on a small set of CSTs, and so it is possible that our observations cannot be generalized when applied to evaluation of other CSTs.

Throughout the study, we collected data on the behavior of users to help explain our findings. We noticed that in rare cases, users abandoned search tasks without waiting to get results and click on them, which could have been due to the slow performance of GRANULATED-LEX possibly affecting normal user behavior. This could have biased the evaluations we performed against the faster SANDO and ENHANCED-SANDO techniques. Some of our developers were providing non-English queries, which were matching code elements that were non-English, which could have also affected our results (see Section IV for a thorough discussion). Note that the evaluation results themselves were not the main goal of the case study as much as learning how to apply paired interleaving for evaluation of CSTs.

VI. RELATED WORK

There has been little work on improving the infrastructure for evaluation of text analysis techniques and their client software engineering tools. A recent comprehensive survey of feature location techniques by Dit et al. [2] supports the motivation for this kind of work, concluding that a major impediment to progress in feature location research is the difficulty in comparing approaches.

The TraceLab [6] research environment's goal is to facilitate rapid experimentation and evaluation of code search. Using TraceLab, CSTs are rapidly constructed by composing a set of TraceLab components, and evaluated using a set of (five) provided gold sets. Because TraceLab is efficient at constructing and evaluating CSTs, we envision that future CST evaluation may combine both paired interleaving and TraceLab, by starting with a TraceLab study as a proof-of-concept, followed by a comprehensive comparative evaluation in the field using paired interleaving.

Sando [7] is a research-extensible local code search tool, which has also been proposed as a means to improve the effectiveness of CST research by improving evaluation and dissemination. As Sando is a practical CST tool intended for developers, it could also be used as a platform for conducting CST evaluation via the paired interleaving approach proposed in this paper.

Researchers in information retrieval have proposed many approaches to predict query difficulty [15]. Some include metrics to estimate the lexical difficulty of the query (e.g., ambiguity and polysemy by extracting features including average number of morphemes per query word, number of proper nouns,...) and some are based on statistical features

[4]This was the average of the short queries available with TraceLab. The long queries that are also provided are roughly one order of magnitude longer.

of the query such as term co-occurrence statistics. The paired interleaving implementation is extensible enough to enable deeper analysis of the queries and their matches with these more sophisticated metrics. We believe the paired interleaving evaluation can be strengthened by leveraging the work on query performance prediction for concept location in the data analysis [16], [17].

There have been a few studies of code search behavior when developers are searching on the web [12]. They find interesting results including that developers use source code on the web for various purposes, use different forms of queries to express their information needs, including natural language and names of code entities they are aware of. We have observed similar variations in queries in our case study which focuses on local code search. Others have focused on studying how programmers decide what to search for and how they decide which results are relevant to their assigned maintenance task in a local code base [13]. Among their findings, they found that users form search queries based on experience and expectations around naming conventions, with searches that were often very general returning many results.

VII. Conclusions

Our case study has demonstrated that even with a small number of developers using such a paired interleaving-based evaluation system during their daily maintenance tasks, we are able to learn how CSTs perform in the field and also identify practical improvements to evolve CST technology. By studying the queries collected during this case study, we observed how different they can be from queries currently used in studies, potentially misguiding researchers during their evaluations of CSTs. We believe that coupling this kind of field comparison of CSTs with current evaluation practices will help advance the state of the art in code search techniques towards accounting for users' behavior during tool use.

References

[1] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. on Soft. Eng.*, vol. 32, no. 12, pp. 971–987, 2006.

[2] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of Soft. Maint. and Evolution: Research and Practice*, 2011.

[3] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Working Conference on Reverse Engineering*, 2004, pp. 214–223.

[4] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Int. Conf. on Software Engineering (ICSE)*, 2011.

[5] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. V. Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *Int. Conf. on Aspect-Oriented Software Development*, 2007.

[6] B. Dit, E. Moritz, and D. Poshyvanyk, "A tracelab-based solution for creating, conducting, and sharing feature location experiments," in *IEEE Int. Conf. on Program Comprehension*, 2011.

[7] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz, "Sando: an extensible local code search framework," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE, 2012, pp. 15:1–15:2.

[8] E. Hill, "Integrating natural language and program structure information to improve software search and exploration," Ph.D. dissertation, University of Delaware, August 2010.

[9] O. Chapelle, T. Joachims, F. Radlinski, and Y. Yue, "Large-scale validation and analysis of interleaved search evaluation," *ACM Trans. on Information Systems*, vol. 30, no. 1, Mar. 2012.

[10] D. Lawrie, H. Feild, and D. Binkley, "Leveraged quality assessment using information retrieval techniques," in *14th International Conference on Program Comprehension*, 2006.

[11] T. Joachims, "Evaluating retrieval performance using clickthrough data," in *Text Mining*, J. Franke, G. Nakhaeizadeh, and I. Renz, Eds. Physica/Springer Verlag, 2003, pp. 79–96.

[12] S. K. Bajracharya and C. V. Lopes, "Analyzing and mining a code search engine usage log," *Empirical Softw. Engg.*, vol. 17, no. 4-5, pp. 424–466, Aug. 2012.

[13] J. Starke, C. Luce, and J. Sillito, "Searching and skimming: An exploratory study," in *IEEE International Conference on Software Maintenance (ICSM*, sept. 2009, pp. 157 –166.

[14] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[15] D. Carmel and E. Yom-Tov, *Estimating the Query Difficulty for Information Retrieval.* Morgan and Claypool, 2010.

[16] S. Haiduc, G. Bavota, R. Oliveto, A. De Lucia, and A. Marcus, "Automatic query performance assessment during the retrieval of software artifacts," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE. New York, NY, USA: ACM, 2012, pp. 90–99.

[17] S. Haiduc, G. Bavota, R. Oliveto, A. Marcus, and A. De Lucia, "Evaluating the specificity of text retrieval queries to support software engineering tasks," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE. Piscataway, NJ, USA: IEEE Press, 2012, pp. 1273–1276.