Precise and Scalable Querying of Syntactical Source Code Patterns
Using Sample Code Snippets and a Database

Oleksandr Panchenko¹, Jan Karstens², Hasso Plattner¹, and Alexander Zeier¹

¹Hasso Plattner Institute, Potsdam, Germany, Email: {panchenko, office-plattner, zeier}@hpi.uni-potsdam.de
²SAP AG, Walldorf, Germany, Email: jan.karstens@sap.com

Abstract—While analyzing a log file of a text-based source code search engine we discovered that developers search for fine-grained syntactical patterns in 36% of queries. Currently, to cope with queries of this kind developers need to use regular expressions, to add redundant terms to the query or to combine searching with other tools provided by the development environment. To improve the expressiveness of the queries, these can be formulated as tree patterns of abstract syntax trees. These search patterns can be expressed by using query languages, such as XPath. However, developers usually do not work with either XPath or with AST. To shield developers from the complexity of query formulation we propose using sample code snippets as queries. The novelty of our approach is the combination of a query language that is very close to the surface programming language and a special database technology to store a large amount of abstract syntax trees. The advantage of this approach over existing source code query languages and search engines is the performance of both query formulation and query execution. This paper describes the technical details of the method and illustrates the value of this approach with performance measures and an industrial controlled experiment. All developers were able to complete the tasks of the experiment faster and more accurately by using our tool (ACS) than by using a text-based search engine. The number of false positives in the result lists was significantly decreased.

Keywords—source code search; query-by-example; source code query language; abstract syntax trees; XPath; database

I. INTRODUCTION

Source code search is an important activity for program comprehension. In some scenarios (e.g., concept location or looking for a component to reuse) it is sufficient to use text-based search engines or source code query languages with a coarse level of granularity. However, while fixing bugs, developers often search for fine-grained syntactical patterns, which consist of just few statements or even a fragment of a statement. An example of such a query is a search for all function invocations which import a parameter named filename or “find all reading accesses to a database table but000” are in essence AST patterns. These two patterns are marked in Figure 1 using an underline and a waved underline respectively.

Such structural queries can be formulated as XPath expressions, because as any XML document, an AST is in fact a tree. The obvious advantage of XPath over text-based search is that XPath can express relations between fine-grained source code elements. However, XPath queries are difficult to formulate, as developers usually do not work with either XPath or with AST.

This paper proposes a query-by-example approach in which the developer formulates a query as a sample code snippet and the corresponding XPath query is automatically generated and executed. A developer needs to write a snippet of code that is structurally similar to the code of interest. Tokens that are not relevant in the context of the search are replaced by placeholders (denoted as “…”) or just omitted. It is also possible to use some of the regular expression patterns, e.g., asterisk or “!”. An example of such a snippet that retrieves all function calls that have a formal parameter with an underline and a waved underline respectively.

source code is as follows:

```
DATA file_util TYPE REF TO cl_f_util.
file_name = 'logfile0'.
CALL FUNCTION f_check
  IMPORTING filename = file_name.
SELECT name_org FROM but000 INTO cstmr.
```

For example, the queries “find all function invocations which import a parameter named filename” or “find all reading accesses to a database table but000” are in essence AST patterns. These two patterns are marked in Figure 1 using an underline and a waved underline respectively.

Our approach works as follows. (1) Source code is parsed and the resulting ASTs are stored into a central database. Because we do not need to use XML format, we store the actual trees without the serialization of these as XML files. (2) Developers formulate their queries as sample code snippets which are parsed into AST fragments and the corresponding XPath expressions describing those fragments are generated automatically. (3) XPath queries are evaluated by a query engine and the result list contains links to source code documents as it would in any other search engine.

Working with a large amount of tree-shaped data implies the use of a database technology. This paper introduces a
special data model that allows no-loss albeit compact storage of a large number of ASTs (Section IV). The tree indexing has enabled the proposed data model to expeditiously answer fine-grained structural queries.

The ideas of querying ASTs using XPath and the use of a database for source code repository are not new and have been separately used in various tools (Section II). However, our experience shows that the use of an off-the-shelf database is not sufficient for large and extra-large repositories. A special database technology proposed in this paper is capable of handling even such amounts of data. This paper illustrates the approach with the help of a controlled experiment in industrial settings (Section VI) and discusses the querying performance based on a real software repository of a large enterprise system (Section V).

To validate the approach we implemented a tool—ACS, which stands for ABAP Code Search. The approach was evaluated using real data. Source code of an enterprise system SAP Business ByDesign\(^2\) was indexed and used for performance measures and the experiment. The version investigated in this paper is currently in the maintenance phase of its lifecycle. The system has 124k development objects, which imply classes, interfaces, function modules, programs, etc. Each development object consists of one or several includes. In total the system has 1.6 Mio includes containing 72 Mio statements. The language of this system is ABAP, a fourth-generation programming language for data processing in commercial applications. In addition to language elements which are usual for general purpose programming languages, ABAP offers functions for background task scheduling, authority checks, operations with database tables, arrays, files, user interface elements, etc., directly as language elements. For example, access of a database table is represented by the \texttt{SELECT} statement. Therefore, the language is rich with more than 800 non-reserved keywords. This results in a large number of types of AST vertices (more than 1100). Nevertheless, the proposed approach does not depend on any particular feature of ABAP and can be applied to any programming language.

This paper makes four contributions:

1. It discovers the fact that while fixing bugs developers often search for fine-grained source code elements and need to specify the relations between them in a search query. This observation was made while analyzing a log file of an existing source code search engine.

2. It proposes to store the source code in the form of ASTs in a special database and to use XPath for querying. ASTs are stored \textit{as is} without any annotation or aggregation, which allows fully automated indexing and working with fine-grained data. Since developers usually do not work with ASTs and prefer query languages that are close to the surface programming language, ACS provides the possibility to write a snippet of code that is structurally similar to the pattern to be found and the ACS will automatically generate a corresponding XPath query.

3. It proposes a database technology that is efficient for storing and querying data of this level of granularity. The technical feasibility of the approach was demonstrated by performance measures.

4. It illustrates the usefulness of the approach by an experiment. In the experiment, eleven developers were asked to complete search tasks by using ACS and by using a text-based search engine. Our approach proved to be more efficient both in terms of time required to complete the task and in terms of accuracy of the results.

II. RELATED WORK

Serialization of source code in an XML-based format has been discussed several times. JavaML \cite{1} is a markup language for AST representation that enables referencing source code elements and transforming source code. srcML \cite{2} provides explicit markup of syntactical information within source code to support various program analysis, fact extraction, and reverse engineering tasks. XSDML \cite{3} uses an XML representation of source code to support manipulation and refactoring. GXL \cite{4} proposes an exchange format to facilitate interoperability of reengineering tools. The idea of JavaML is very close to our approach. However, until now JavaML has not been used together with a database, which limits its applicability to only small local code repositories. ACS is capable of querying large source code repositories simultaneously and interactively. Moreover, JavaML requires

\footnote{2http://www.sap.com/sme/solutions/}
developers to learn a query language, e.g., XPath or XSDoc. The semantic of the underlying tree format is not within the scope of this paper. This means that ASTs produced by a parser are stored as is without any change, transformation or annotation.

The idea of querying AST as a tree is not new. Besides XML-based approaches discussed above, a bunch of research prototypes and industrial tools have been developed based on non-XML representation of ASTs. One of the first successful tools was GENOA [5], which specifies how ASTs should be traversed to answer developers’ questions similar to our scenarios (Section III). Complex algorithms for language-independent AST analysis including recursion were possible. Despite being very similar to our idea, GENOA has two major disadvantages: the query evaluation is slow because of expensive tree traversal operations and it requires learning of the complex GENOA query language. Crew proposed to locate syntactical patterns in ASTs using ASTLOG [6], a Prolog-like query language. Although being flexible, the poor efficiency disallows using this tool interactively. Even given factor 200 in increase of computing power in the last years, ASTLOG queries remain too slow for large code repositories. YAAB [7] uses Object Constraint Language to express patterns of ASTs to be found. The database of the Java Tools Language [8] relies on the relational model, however, it can also represent ASTs. PMD [9] allows formulating the rules as XPath queries on ASTs. Although PMD is without a doubt a useful tool, it is intended to be used as a background job that checks development guidelines and not as an interactive search engine. JQuery [10] is a tool supporting exploration of source code. A flexibility of the proposed views is achieved by a query-based customization of the content presented in the views. Van Dijk et al. proposed a framework for architecture conformance checking [11]. The conformance rules are formulated using Xlinkit, which is a first order language that references XML elements using XPath. A number of other tools exist that use AST for source code querying: TXL [12], Reprise [13], XForm [14], TAWK [15], PQL [16].

The question of what query language should be used for querying source code has been raised many times. The language depends on the scenario in which it is used. We believe that the query language that is close to the surface programming language is efficient for querying of syntactical patterns. The idea of language-oriented pattern matching goes back to the programmer’s apprentice [17]. We would like to mention SCRUPLE tool [18] for matching syntactical patterns and CCEL [19]. Although the syntax of Datalog significantly differs from the syntax of the surface programming language, it is intended to solve the problem of scalability [20]. Although the importance of a query language that is close to the surface programming language has been discussed several times, to our best knowledge no tool has implemented such a concept.

The first key differentiator of our idea is the use of high-performance database technology. Existing tools fail to combine fine-grained search with high querying performance on large source code repositories. The performance measures reported until now [8] do not allow using such tools interactively on large code bases. The data model presented in our paper enables a highly interactive source code querying by preserving the original level of granularity. ACS does not allow free traversal of AST vertices. Of course, it limits the number of possible queries, but we prefer to enable easy-to-use and scalable search facilities, leaving some of the developers’ questions out of scope.

The second differentiator is the ability to query ASTs using just code snippets. One important requirement is the simplicity of the interface for developers. To our best knowledge, the existing AST-based tools require developers to learn a query language, which deviates from the programming language because it tries to cover other artifacts, e.g., call graph or dataflow graph. Although dataflow is a very helpful property for source code analysis, our approach is limited to syntactical code patterns and does not cover dataflow analysis. The advantage we gain because of this limitation is a simple interface for developers and the ability to scale up to hundreds of millions of statements.

### III. Eight Test Scenarios

To select test scenarios we interviewed four developers who maintain the system described in Section I. They reported that the typical search target while fixing bugs is a statement. They often look for usage of an identifier in a specific context and add syntactical constraints to the query. Interesting that for tasks of this kind the ranking of search results is not important because developers are usually interested in all results.

To validate the input of interviewees, we analyzed log files of an existing source code search engine. This search engine is based on an inverted index technique and has been used by the developers for two and the half years. Besides the text-based search the engine supports regular expressions. We extracted part of the log file which is related to the analyzed system. This part contains 812k entries and has 172k distinct queries. As the system is currently in the maintenance phase of its lifecycle we assume that the main performed tasks were fixing bugs and refactoring. The queries were automatically categorized into one of the categories presented in Figure 2. The categorization was done using naming conventions. For example, queries starting with "get_" or "show_" were assigned to the Method name category. The Structural pattern category includes queries where developers used symbols, such as "<", ">", "="", "==", ",", ",", to express the relations between terms of the query. Queries containing ABAP keywords were categorized into the ABAP keyword category. Obviously, a query which contains an ABAP keyword and an identifier
implies some relation between them. The category Other identifier contains local and global variables, parameters, table names, structures and other identifiers. Figure 2 shows that in addition to identifiers, developers often use syntactical patterns and ABAP keywords. In total, the queries that require the ability to express the AST-level relations between terms of the query constitute at least 36% of the log file entries. This number in reality is higher because some of the queries could not be automatically categorized and reside in the category Undefined.

Although it is difficult to make accurate guesses regarding the developers’ original intention behind each query, a manual investigation of queries from the log file showed that most queries can be formulated as code snippets more precisely than as text queries. In the rest of the paper we use the following eight scenarios which both occurred often in the log file and were selected by the interviewed developers.

(1) a search for compute statements that assign a value to a global variable.
(2) a search for an updating access to a database table.
(3) a search for a violation of some development guidelines. For example, all SELECT statements without a WHERE clause should be found, because those statements can result in poor performance. Such queries are typically run during nightly builds. However, if we could answer those queries in real time, why not offer this possibility to developers?
(4) is a modification of (3) with a table name added.
(5) a search for all function invocations which use, e.g., filename, as a formal parameter name. Questions of this type occur during manual refactoring or preventive maintenance. Especially in languages that allow functions to have optional or named parameters, such queries can be used to investigate which parameters are actually used by callers.
(6) a search for all occurrences of a system variable, e.g., uname3, in an ASSERT statement. This query is used to check if the coding contains user-specific testing elements.
(7) a search for data declarations of a certain type.
(8) a search for authorization checks. ABAP features a notion of authorization objects, which describe authorizations to execute certain programs. Authorization objects have fields to carry additional information. Each user can be assigned to several authorization objects describing his role in the system. In the code developers need to check if the program is allowed to be executed with a certain authorization object, for example, using a statement, such as AUTHORITY-CHECK OBJECT 'f_bkpf_koa' ID 'koart' FIELD 's'.

The generated XPath queries for the selected scenarios are discussed in Section V-D.

One can argue that the performance of a source code search engine is not important, since developers should search inside a small code area, otherwise it would indicate pure system modularization. However, many scenarios require searching in the entire code base. Particularly, during refactoring it is important to find all occurrences. While searching for a code example it is difficult to meaningfully reduce the scope, because developers seldom know where to start searching. Our analysis of the log file showed that more than 13% of queries were run against all systems simultaneously (that is, more than $1.4 \times 10^9$ statements!).

IV. DATA MODEL FOR STORING ASTS

An AST-based code representation is not compact, e.g., the size of converted XML documents is about 10 times larger than the original source code [3]. The experiment, which is presented in this paper, confirms this ratio: the serialized documents are about 29 GB, while the corresponding size of original source code is about 3.2 GB. Therefore, a database support is required to handle such an amount of data properly and to enable fast processing of queries.

The adjacency list is a crude approach for storing structural information represented as a tree in a table, each row of which corresponds to a vertex with the reference to its parent vertex. The example of a data model for ASTs stored as an adjacency table is represented in Table I. Attribute VERTEX_ID identifies an AST vertex and is unique within an AST. Since development object names are unique, VERTEX_ID and DEV_OBJECT_NAME build a primary key. Attribute VERTEX_VALUE contains the type of AST vertex (in the case of statement, compound statement, clause or operator) or the actual value of the vertex (in the case of identifier or literal). That is, in our model identifiers and literals are not attributes but vertices. We applied such a de-normalization to improve performance and to save space. Since values of all types of AST vertices are stored in the attribute VERTEX_VALUE, they can be referenced in an XPath query using the same semantic:

```
//CALL_FUNCTION/NAME/check
```

However, it might be the case that some identifiers or literals overlap with AST vertex type names, for example,
CALL_FUNCTION or NAME can be used as identifiers. To avoid this ambiguity, a set of namespaces is used to express the role of the vertex. In our case, \(s\) means statement, \(c\) means compound statement, \(v\) stands for clause, \(opt\) means operator, \(idf\) denotes identifier, and \(lit\) stands for literal (compare Figure 1). Then this query is formulated as:

//\(s:\)CALL_FUNCTION/\(v:\)NAME/idf:\)check

Attribute VERTEXCATEGORY defines the namespace of an AST vertex. Attributes INCLUDE_NAME and SOURCE_POSITION refer to the place in source code which corresponds to the AST vertex. Since an include can be included in several development objects, it is not sufficient just to store development object name, but include name should be stored too. Attribute PARENT_ID refers to the parent vertex. Since AST edges are not labeled, they do not require a special table in which to be stored.

Some improved approaches proposed schema-aware optimizations for storage and querying [21], [22]. Nevertheless, ASTs have a large number of distinct vertex values and the schema of ASTs is too complex to make use of those optimizations.

The non-optimal space usage limits the applicability of this simple data model to large structures. Moreover, storing connected elements in one table results in multiple self-joins, when it comes to traversing the connections, and thus in the querying performance problems. The following data model overcomes both challenges.

The first improvement is the reduction of space requirements by the utilization of compression algorithms. The values in columns of the adjacency table repeat multiple times because the number of identifiers, literals and AST vertex types is finite and each vertex value occurs many times. Therefore, the application of compression algorithms for columns is reasonable (see Table II). Although some attributes have type String, the actual table tuples contain only compressed value IDs. For each attribute a dictionary is maintained to map the value ID with the actual attribute value. The approach of having a separate table for the values has been used for native XML databases as well [21]. The size of the column includes not only the value IDs, but also a dictionary and an index which enables faster table scans. In the proposed approach both dictionary encoding and run-length encoding [23] are used.

The second contribution is the performance improvement through the use of an indexing algorithm. To overcome the performance challenge caused by multiple self-joins, ACS uses the pre-post-order numbering method [24]. Two numbers are assigned to each vertex in the tree: a pre-order number and a post-order number. These numbers are ascertained during a depth-first traversal of the tree before and after the visit to the corresponding vertex (see Figure 1). If vertex \(B\) is a descendant of vertex \(A\), \(B\) must have a higher pre-order and lower post-order number than \(A\). Compare vertices \(v:\)IMPORTING and \(idf:\)filename in Figure 1. This indexing accelerates joining AST vertices along all tree axes [24]. Now it is possible to retrieve all descendants of a vertex with just one table scan. Since the attribute PRE_ORDER is unique inside each AST, this attribute can be used as a key, e.g., for parent-child relation in the attribute PARENT_PRE_ORDER.

V. Feasibility Evaluation

The research question addressed in this section is: is it technically feasible to answer developers’ questions formulated as code snippets on a large portion of source code? The proposed approach was evaluated in terms of required space, indexing and querying performance, and query formulation accuracy.

A. Implementation

ACS consists of four parts: a crawler, an AST table, a query engine, and a search client (s. Figure 3). The crawler traverses source code, constructs ASTs, indexes them, and stores the data in the AST table. The SAP application server has been used to parse ABAP source code.

The database layer stores the AST table. Numerous XML databases exist. There are two categories of XML databases in respect to the storage format: document-based storage, and vertex-based storage. The former ones store entire documents as a text into a database and provide indices over the document to accelerate retrieving. The latter ones store individual vertices and the relations between them into a special database schema. Obviously, document-based XML databases can return entire documents or document fragments. However, in our scenario merely a list of references to the source code documents should be returned. Therefore, application of a vertex-based database seems reasonable. Various mapping schemas have been developed for vertex-based databases [21]. MonetDB\(^4\) offers a mapping schema that is most similar to what is proposed in this paper. Nevertheless, we decided to implement ACS including query engine on top of TREX\(^5\) because its compression capabilities fit our requirements and because TREX offers an integrated text search engine which is used to store the textual representation of source code. This means that a text-based

\(^4\)http://monetdb.cwi.nl
\(^5\)http://www.sdq.sap.com/frj/sdn/nn-enterprisesearch
Table II
STRUCTURE OF THE AST TABLE

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th># of Rows</th>
<th>Cardinality</th>
<th>Bits</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK DEV_OBJECT_NAME</td>
<td>String</td>
<td>458,030,329</td>
<td>123,963</td>
<td>17</td>
<td>952</td>
</tr>
<tr>
<td>VERTEX_VALUE</td>
<td>String</td>
<td>458,030,329</td>
<td>5,669,440</td>
<td>23</td>
<td>1,340</td>
</tr>
<tr>
<td>VERTEXCATEGORY</td>
<td>String</td>
<td>458,030,329</td>
<td>6</td>
<td>3</td>
<td>168</td>
</tr>
<tr>
<td>PK PRE_ORDER</td>
<td>Integer</td>
<td>458,030,329</td>
<td>5,338,620</td>
<td>23</td>
<td>1,307</td>
</tr>
<tr>
<td>POST_ORDER</td>
<td>Integer</td>
<td>458,030,329</td>
<td>5,338,270</td>
<td>23</td>
<td>1,307</td>
</tr>
<tr>
<td>INCLUDE_NAME</td>
<td>String</td>
<td>458,030,329</td>
<td>1,614,445</td>
<td>21</td>
<td>1,185</td>
</tr>
<tr>
<td>SOURCE_POSITION</td>
<td>Integer</td>
<td>458,030,329</td>
<td>115,806</td>
<td>17</td>
<td>951</td>
</tr>
<tr>
<td>FK PARENT_PRE_ORDER</td>
<td>Integer</td>
<td>458,030,329</td>
<td>3,368,863</td>
<td>22</td>
<td>1,243</td>
</tr>
</tbody>
</table>

search is supported as well. To do so we extended TREX with a query engine, which is aware of the optimizations of our approach, that is, namespaces, storing vertex types and values in one table field, and pre-post-order numbering. As shown in Subsection V-D, the overhead of the implementation is paid off by better performance. It is also possible to parallelize indexing and querying, to partition the AST table horizontally based on packages or versions, and to distribute the table between several servers. In this way, the scalability of the proposed approach is ensured. We also applied some database optimizations to accelerate the queries. For example, we let TREX keep tuples physically sorted by DEV_OBJECT_NAME and secondary-sorted by PRE_ORDER. It allowed us to use sort-merge join algorithm (Subsection V-D). Note that ACS query engine does not fully support the XPath specification, though is able to answer most of the targeted queries.

The SAP development environment does not provide a local source code storage for developers, rather all source code is stored centrally. All code modifications are done directly in the central code repository. The AST table is stored on a central server for two reasons: (1) it is too big to be stored locally, but in many cases developers want to search globally; (2) changes in the central code repository can be replicated in the AST table immediately.

The query engine was implemented in C++ and integrated directly into the TREX server as an extension. The search client including XPath query generator was implemented as an Eclipse plugin on the client side and is integrated into the development environment.

B. Source Code vs. XML Benchmarks

Although XML benchmarks are naturally used for performance measurement, we use our own test data because existing benchmarks differ from source code ASTs in a number of properties. Figure 4 illustrates these properties in the example of XBench [25] and XMark [26]. Each benchmark provides a tool for data generation to create data files of the size required. With XMark it is possible to generate XML files of comparable size (Figure 4, part A) and depth (Figure 4, part B) to source code ASTs. Nevertheless, the structure of the generated trees remains too simple. XBench has only 26 distinct types of vertices, XMark has 74 types, but, Java has 108 types of AST vertices and ABAP has even more than 1100. Thus, the existing benchmarks are limited due to little diversity of vertex types. But the most important difference is that source code has a much higher structural complexity: while XBench and XMark prescribe how the vertices should be composed into a tree, AST vertices have much more composition possibilities. Many vertices can be used in different contexts, and, hence, can have parent vertices of various types. The distribution of possible parent vertices is presented in Figure 4, part C.

We decided to use real data for the performance measurement instead of using an existing benchmark because the properties of data have a great impact on query formulation, query optimization and storage optimization.

For the test data the size of the table compressed using dictionary encoding is 458k rows * 149 bits per row =
9.2 GB. For more details see Table II. Although run-length encoding allows saving some space, extra space is required for database indices and dictionaries. In total the database requires 8.5 GB, which is reasonable for a repository of this size and granularity. In comparison, the size of original source code in text files is 3.2 GB.

C. Indexing Performance Evaluation

The indexing process consists of the following steps: (1) parsing of source code into ASTs. This step is done using ABAP parser which is not a part of the ACS presented in this paper. (2) preprocessing prepares structural data for storage in a database. This step includes the pre-post-order numbering. (3) actual storing of the data into the database. This step requires a significant amount of time because of compression. On two Xeon® E5450 processors the compression was done with a throughput of 5,000 rows per second. All three steps can be parallelized.

Since the source code is stored on a central server, as soon as the changed code is checked in the central code repository, the AST table is updated respectively. We observed the size and frequency of commits in the indexed system and discovered that on average only 4% of development objects were changed in a two week period. Therefore, the proposed approach enables real-time update of the table.

D. Querying Performance Evaluation

This section illustrates the performance of the eight queries which correspond to the scenarios given in Section III. The sample code snippets and corresponding XPath queries are given in Table III.

The query engine parses the XPath expression into a query tree which consists of table scans (σ) and operations (joins, unions, intersections, inversions, etc.) on the results of the table scans. A simplified query tree for query #6 is presented in Figure 5. Each table scan is issued independently, and operations on the results of the table scans are then applied. The performance measures are presented in Table III in the column Execution time. An investigation of performance showed that query execution time depends on the total number of result rows in table scans. This is no surprise because the selectivity of the table scans influences the number of comparisons completed in the join operators. This means that queries with high selective table scans, e.g., #2, have a shorter query execution time than queries with low selective table scans, e.g., #1. This is illustrated in Figure 6. Each of 30 measures presents an average of five runs. The query execution time can be predicted just by looking at the query. To do so we only need to know the sizes of all possible table scans. This information can be easily obtained from the frequency distribution of the AST vertex types.

The obtained performance results based on a MonetDB database are also presented in Figure 6. More detailed performance comparison and also comparison with other databases is excluded here to save space. A better performance of ACS is achieved mainly because of the domain knowledge. For example, we were able to better parallelize join computation because we know that results can be matched only inside of a single development object, thus, we split the results of the table scans by DEV_OBJECT_NAME.

Comparing these performance results with previous studies [27], we have achieved a better performance on a larger code base. Our code base is 10 times larger than Eclipse in terms of the number of ASTs, 130 times larger in terms of the number of files/includes, and 90 times larger in terms of lines of code. Despite the fact that both studies are not directly comparable because of differences in size, queries and hardware (two Xeon® E5450 instead of one Pentium® IV 3.2 GHz/HT), we believe that the only possible way of analyzing a large source code repository is by using a specialized in-memory database system.

Traditionally, performance measures for source code querying have been concerned with the problem of reducing query execution time. However, in an environment where numerous developers send their questions to a central repository, another question is more important: how do we get a fixed (low) response time as cost-effective as possible? Figure 7 illustrates that on an eight-core server throughput decreases if more than eight users run queries in parallel.

Assuming average ABAP statement to correspond to two lines of code.
Table III

<table>
<thead>
<tr>
<th>#</th>
<th>Query</th>
<th>XPath query (simplified)</th>
<th>∑ size of σ (total)</th>
<th>Execution time</th>
<th>Size of Π (selected packages)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>gv_valueguid = ...</code></td>
<td>`//s:COMPUTE</td>
<td></td>
<td>s:MOVE`</td>
<td>17,165,571</td>
</tr>
<tr>
<td>2</td>
<td><code>UPDATE but000</code></td>
<td><code>//s:UPDATE</code></td>
<td>251,650</td>
<td>0.35s</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>`SELECT ... !WHERE</td>
<td>`//s:SELECT</td>
<td></td>
<td>s:SELECT_LOOP`</td>
<td>1,190,458</td>
</tr>
<tr>
<td>4</td>
<td><code>FROM but000 !WHERE</code></td>
<td>`//s:SELECT</td>
<td></td>
<td>s:SELECT_LOOP`</td>
<td>1,191,734</td>
</tr>
<tr>
<td>5</td>
<td><code>CALL FUNCTION ... IMPORTING f=name</code></td>
<td>`//s:CALL_FUNCTION</td>
<td>5,484,262</td>
<td>1.82s</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td><code>ASSERT name</code></td>
<td>`//s:ASSERT</td>
<td></td>
<td>s:ASSERT_SHORT`</td>
<td>150,373</td>
</tr>
<tr>
<td>7</td>
<td><code>DATA ... TYPE REF TO cl_f_util</code></td>
<td><code>//s:DATA_DEFINITION</code></td>
<td>21,088,838</td>
<td>3.92s</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td><code>'f_bkpf_koa' ID 'koart' FIELD 's'</code></td>
<td><code>//s:OBJECT/lit:f_bkpf_koa</code></td>
<td>2,347,244</td>
<td>1.21s</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 7. Throughput vs. Average Query Execution Time

E. Query Formulation Accuracy

Besides the performance of query execution, an important factor is the accuracy of query formulation. Since XPath is a structured query language, the usual characteristics for search engines, such as recall or precision, are not applicable. As soon as a query is formulated correctly, the result set is evaluated accurately. Therefore, the crucial step is the query formulation. The snippet entered by the developer is parsed by the search client. The AST of the snippet should not be necessary 100% correct. We achieved quite good results just by composing the query of the following type: `//s:<statement_vertex>/idf:<identifier>`

The parser was a light-weight stateful top-down parser with several branches for each type of snippet structure: with or without leading keyword, identifier only, assignment statement, etc. Often, a snippet could be translated into several alternative XPath expressions, e.g., both `//s:COMPUTE` and `//s:MOVE` correspond to an assignment statement. To evaluate the recall of the XPath query generator we randomly selected 200 statements from the existing source code. We randomly modified these with the fulfillment of the following conditions: (1) the beginning of the statement (the main keyword) remained unchanged, (2) some of the clauses were removed, (3) some of the identifiers were replaced by placeholders. After that we automatically translated those snippets into the XPath queries and ran the queries. In about 90% of cases the result of the XPath query contained the link to the original include from which the snippet was selected to generate the query. We also tried to generate XPath queries from snippets that contained two or more statements. But in this case the number of successful generations decreased because the queries became more complex, error-prone and also resulted in long execution times. To express the relations between the statements some additional rules would be required, therefore, we reduced the scope of our tool to a fragment of a single statement. The source of generation errors for single statement snippets was the fact that the ABAP keywords are not reserved and can be used as identifiers. That is, the statement `DATA TYPE ...` is a correct ABAP statement. Since we were not able to use the fully-featured ABAP parser for query generation, such cases resulted in wrong XPath queries. Nevertheless, such misuse of keywords occurs seldom. Some problems caused ABAP macros, because their definitions may contain not syntactically correct code fragments. Some clauses are not mapped to any keyword and, thus, cannot be expressed using query language. This negatively influences the precision. We guess that in more simple languages with a stricter syntax the query generation will be more precise.

VI. CONTROLLED EXPERIMENT

The research questions investigated in this experiment are: (1) is it possible to express developers’ questions as sample code snippets? are developers able to formulate the queries
efficiently? (2) does the approach provide any time saving in comparison to the text-based search, regular expressions and the use of where-used lists? (3) based on the results, which types of tasks can benefit most from the use of ACS?

We decided to compare ACS to a text-based search engine for three reasons: firstly, it provides a similar performance on large code bases; secondly, it features a simple interface; and thirdly, although it has not been able to answer structural queries, it works on fine-grained tokens and disregards higher-level semantic, such as dataflow graphs.

Eleven developers with work experience between two and seven years were asked to go through each of eight scenarios described in Section III. Before the test they had been explained the rationale behind XPath expressions on ASTs, and given a demo of the possibility of querying using code snippets. The training was completed individually and took around 20 minutes. The developers had no previous experience with XPath or ASTs, but had used search engines in their daily work. All developers participated on a voluntary basis and can therefore be assumed to be properly motivated. Each developer completed four random tasks (scenarios) by using ACS and the remaining four tasks by using a text-based search engine. The tasks were completed in arbitrary order. The correctness and the completeness of each task fulfillment were examined after the task was finished. Although the performance measures in the previous section were done on the complete dataset, we selected several packages for both tools to ease the tasks for the developers. The column Size of II in Table III depicts the number of hits inside the selected packages only.

During the test the screens of the development environment were captured on video. Table IV summarizes the results of the experiment. Both tools require nearly the same number of queries to fulfill the task. Nonetheless, ACS features a better total time required. The reason for that is that a text-based search engine requires a significantly higher number of development objects from the result list to be examined because of a high number of false positives. All developers failed to complete task #3 using the text-based search engine, but were able to complete all tasks using ACS. Notably, queries with a NOT condition and queries including often used identifiers gained in precision. The averages were calculated for data except query #3. The number of false positives (# of dev. objects examined – Size of II) was significantly decreased.

VII. Threats to Validity

Although we validated the major aspects of our approach in a large-scale industrial environment, some limitations still apply. Firstly, we validated the approach applied to only one language. The question about the efficiency of the approach in other systems/languages remains open. We believe that the implementation of the approach for simpler languages with a stricter syntax will be easier. However, it should be empirically proven if developers’ efficiency can be improved. Secondly, the experiment presented in this paper is not statistically significant to predict time saving due to use of ACS. Finally, we did not compare the usability of query formulation using ACS with other AST-based source code query languages. Although the advantage is obvious, such a comparison would make the difference tangible.

VIII. Conclusion

This paper demonstrates the possibility of creating a compact yet fine-grained database representation of source code for search engines. Abstract syntax trees obtained by parsing of source code were stored in a special database. An interface based on code snippets was developed to enable high-performance querying ASTs. The experiment conducted showed that the use of source code repositories of this type can significantly accelerate the process of source code search and improve the search accuracy. While many other research efforts in this field imply filling source code repository with some high-level facts extracted from source code, we propose storing fine-grained data without aggregation, as is. Our interviews with developers showed their dislike of formulating long queries and their acceptance of having a reasonably small number of false positives in the result list. The analysis of the log file confirmed it, because an average query has only 18 characters. Thus, our focus lies rather on simple queries instead of covering a broad set of scenarios. The results presented in this paper demonstrated the feasibility of storing and retrieving a large amount of fine-grained structural data. Despite a large data volume the use of the special database allows an interactive use of the repository.

Acknowledgement

This work was done in close cooperation between the Hasso Plattner Institute for Software Systems Engineering and SAP AG. In particular, we thank Olga Mordvinova, Wolfgang Stephan, Arian Treffer, Wester Wei, Sebastian...
Wolfgang, and Xiwei Zhou for their contribution to ACS. We thank the developers who participated in the experiment.

REFERENCES


