Expressive and Scalable Source Code Queries with Graph Databases

Raoul-Gabriel Urma
Computer Laboratory, University of Cambridge
raoul.urma@cl.cam.ac.uk

Alan Mycroft
Computer Laboratory, University of Cambridge
alan.mycroft@cl.cam.ac.uk

Abstract

Source code querying tools are of growing importance. They help software engineers explore a system, or find code in need of refactoring as coding standards evolve. They also enable languages designers to understand the practical uses of language features and idioms over a software corpus.

Existing systems are implemented on top of relational or deductive databases and make ad-hoc tradeoffs between scalability, amount of source-code detail held in the database, and expressiveness of queries.

We argue both that a graph data model is a better conceptual fit for querying source code and also that graph databases provide an efficient scalable implementation even when storing full source-code detail. We show that a graph query language can naturally express queries over source code. Moreover, graph databases, with their emphasis on relations, support *overlays* allowing queries to be posed at a mixture of syntax-tree, type, control-flow-graph or dataflow levels.

We describe and evaluate our prototype source-code query system built on top of Neo4j using its CYPHER graph query language. Experiments confirm that our system scales to programs with millions of lines of code while also storing full source-code detail.

Categories and Subject Descriptors D.2.0 [*Software Engineering*]; D.3.0 [*Programming Languages*]

General Terms Languages

Keywords source code, query, graph, database

1. Introduction

Large programs are difficult to maintain. Different modules are written by different software engineers with different programming styles. However, enhancements to a system often require modifying several parts at the same time. As a consequence, software engineers spend a large amount of their time understanding the system they are working on.

Various automated tools have been developed to assist programmers to analyse their programs. For example, several code browsers have been developed to help program comprehension through hyperlinked code and simple queries such as viewing the type hierarchy of a class, or finding the declaration of a method. These facilities are nowadays available in mainstreams IDEs but are limited to fixed query forms and lack flexibility.

Recently, several source-code querying tools have been developed to provide flexible analysis of source code [1, 5, 15, 17, 20, 21]. They let programmers compose queries written in a domain specific language to locate code of interest. For instance, they can be used to enforce coding standards (e.g. do not have empty blocks) [7], locate potential bugs (e.g. a method returning a reference type `Boolean` and that explicitly returns `null`) [3], code to refactor (e.g. use of deprecated features), or simply to explore the codebase (e.g. method call hierarchy).

Such tools are also useful for programming language designers to perform program analysis on corpora of software to learn whether a feature is prevalent enough to influence the evolution of the language. For example, a recent empirical study made use of a source-code query language to investigate the use of overloading in Java programs [19].

Typically, source-code querying tools make use of a relational or deductive database to store information about the program because it is inconvenient to store information about large programs in main memory. However, they typically let users query only a subset of the source-code information (e.g. class and method declarations but not method bodies) because storing and exposing further information affects the scalability of the querying system. As a result, existing querying facilities have different trade-offs. We distinguish between two categories. On one hand, some systems scale to programs with millions of lines of code but provide limited information about the source code. For example, statement level information is discarded. On the other

OOPSLA'13, Draft

Reprinted from OOPSLA'13,, [Unknown Proceedings], Draft, pp. 1–13.

hand, some systems store more information about the source code but do not scale very well to large programs.

In this paper, we argue that it is possible to build a system that overcomes these tradeoffs.

We observe that programmers are familiar with compiler-like graph structures such as the parse tree, control flow graph and type hierarchy. We therefore suggest that the *graph data model* [12] is a natural conceptual fit for representing source-code information. We present a model for Java source code that can be queried for expressive queries involving full structural, type and flow information. For example, we can query statement level information, the actual type of any expression, the type hierarchy, the call graph of any method, read/write dependencies of variables and fields as well as liveness information. In addition, we explain how our model can be easily extended via overlays and queried using graph traversal operations.

We implemented a source-code querying system based on this model using a *graph database* (i.e a database specialised for storing and traversing graph-like structures). Experiments show that our prototype based on Neo4j [9] scales to programs with millions of lines of code and compares well to existing work.

The main contributions of this paper are as follows:

- The identification of different views of a program that are useful to query for common software engineering tasks and language design questions (Section 3). We make these available as graph *overlays* (Section 4).
- A model for representing Java source-code information based on the graph data model that can support structural, type and flow queries (Section 4 and 5).
- An implementation of a source-code query system based on this model using a graph database (Section 6).
- A detailed set of experiments that demonstrates that our prototype is expressive and scales with increased program sizes (Section 7).

2. Background

In this section, we describe the *graph data model* and *graph databases*.

2.1 Graph Data Model

The graph data model encodes entities and relationships amongst them using a *directed graph structure* [12]. It consists of sets of nodes and edges, where nodes represent entities and the edges represent binary[1] relationships between these entities. Nodes and edges can be labelled, typically with the name of the entity they represent, but possibly with a tuple of name and further values, such as "weight:3". For our purpose, we will refer to such tuples as *properties* on nodes or edges.

As an example, a node may have a property `nodeType` holding the string/enumeration value `JCBinary` to represent a binary expression. This node would typically be connected to two other nodes via two distinct edges labelled `LEFT` and `RIGHT` to connect the left and right child nodes of a binary expression.

By contrast, the *relational model* represents entities as tuples that are grouped into formal relations. Relationships between entities are encoded by combining relations based on a common attribute.

2.2 Graph Databases

Databases supporting the graph data model are often referred to as *graph databases*. They can be accessed via queries expressing graph operations such as traversal, pattern matching and graph metrics. In Section 5 we discuss a graph query language called *CYPHER*, which is used by the Neo4j graph database.

By contrast databases supporting the relational model are referred to as *relational databases*. They are typically accessed using SQL as the query language.

There are two fundamental differences between relational and graph databases which we exploit in our work: *index-free adjacency* and *semi-structured data*.

First, graph databases provide index-free adjacency. In other words, records themselves contain direct pointers to list of connected nodes. This obviates the need to combine relations based on a common attribute to retrieve connected records. Vicknair et al. evaluated the performance of a graph database against a relational database [31]. They report that graph databases executed better for traversing connected data, sometimes by a factor of ten.

Second, relational databases depend on a schema to structure the data whereas graph databases typically do not have this requirement (i.e. they support *semi-structured data*). In practice, it means that in a graph database all records need to be examined to determine their structures, while this is known ahead of time in a relational database. This potential source of inefficiency is addressed in some graph databases which provide index facilities to map certain structures known ahead of time to a list of records. In addition, because graph databases do not depend on a schema, they can be more flexible when the structure of records needs to be changed. In fact, additional properties can be added to or removed from records seamlessly, which facilitates the addition of user-defined overlays. In relational databases, a modification of the schema is required before altering the structure of records.

3. Program Views

In this section, we identify various syntactic and semantic properties of programs which are useful in searches. We

[1] In principle the formulation allows hyper-graphs with hyperedges of having more than two nodes, but we only find the need to use ordinary directed edges.

make these available as *overlays* in the graph model (see Section 5) so we can form queries combining various concepts, such as "find a local variable which is declared in a method called recursively and which has multiple live ranges". These overlay properties can be seen as database *views*. For each property, we discuss their use for common software engineering tasks and language design questions.

- **Text/lexical**
  These queries express grep-like queries to match pattern on the source code. They have an expressive power limited to regular expressions. Such queries are useful to check certain coding standards (e.g. check whether variable declarations follow a naming convention) and also for *program comprehension* (e.g. track certain identifiers in the code base). There exist many regular-expression-based code search facilities such as Google code search [4].
- **Structure**
  Structural structural queries operate on the abstract syntax tree (AST). These queries can be classified into two categories: *global structure* (e.g. querying for abstract method declarations within a specific class) and *statement-level* (e.g. querying for a chain of `if` statements to refactor into a `switch`) [24].
- **Semantic Hierarchy**
  These queries explore semantic hierarchies among program entities. For example, a query language could provide support for the overriding or implementation hierarchy of methods. In addition, it could provide support for the type hierarchy defined in the code base. This is useful both for *program comprehension* (e.g. to find all concrete classes of an interface) and *optimisation* (e.g. perform class hierarchy analysis and eliminate dynamic dispatches).
- **Type**
  Here we effectively query a type-decorated AST of the program. For example, in the context of *program comprehension* one could query method calls on a variable that has a particular type. Such a query requires the availability of the declared types of variables. In addition, some *coding standards* require type information (e.g. "avoid using arrays covariantly").
- **Control Flow**
  Control flow queries investigate the possible execution paths of a program. Such information can be useful in the context of *program comprehension*. For instance, to get a high-level overview of an algorithm or to compute software metrics such as the cyclomatic complexity. In addition, one may wish to explore a codebase by querying for all method calls made inside a program entry point (e.g. `main()`) or to analyse which parts of a module are calling methods from a database API. Such query requires access to the method call graph of a program.
- **Data Flow**
  Data flow queries examine the flow of values in a program. For example, one could query whether a variable is used in the future by inspecting the liveness set of each statement. This is useful for *optimisation* and for removing unused variables. In addition, in the context of *program comprehension* one could query whether a method reads or write a specific field, the reaching definition of an assignment, or the number of items a variable may point to.
- **Compiler Diagnostics**
  It is sometimes useful to access extra information generated by the compiler. For example, one may want to query the list of warnings associated with a specific block of code. In addition, in the context of *program comprehension* it could be useful to query various decisions made by two versions of a compiler to identify incompatibilities. For example, Java 8 brings more aggressive type inference compared to Java 7. There has been discussion about how to identify possible incompatibilities by inspecting the inference resolutions log generated by the Java compiler [6].
- **Run-time Information**
  Run-time information is useful in order to investigate the impact of code at run time such as its memory consumption and CPU usage. For example, synchronized blocks are often responsible for performance bottlenecks. In fact, there are query technologies that can analyse Java heaps to facilitate application troubleshooting [8].
- **Version History**
  Certain type of queries can only be answered by analysing the version history of a codebase. For example, in the context of *program comprehension* it could be useful to query for methods that have been updated the most, to find all modifications made by a specific committer or simply to perform code diffs [22]. Such information is also useful for *programming language research* to investigate the adoption of certain development practices, new libraries or language features over time [23]. In addition, recent work suggested the use of version history to mine aspects from source code [14].

4. A Graph Model for Source Code

Our model to represent source-code information consists of several compiler-like graph structures. We enhance these structures with additional relations, which we call *overlays*, to enable and facilitate queries over multiple program views.

In our model, source-code elements are represented as nodes and the relationships amongst them as edges. We focus on Java. However, our methodology can be applied to any object-oriented language. The design of our model was guided by our motivation to support full structural informa-

tion about the source code as well as type and flow information.

We provide some examples on how to query our model. Data manipulation is explained in more detail in Section 5.

4.1 Abstract Syntax Tree

The AST of a program is a directed acyclic graph where the nodes represent source-code entities and the edges represent directed parent to child relationships. The AST fits therefore naturally with the graph data model.

The Java compiler defines different types of AST node types to represent different source-code elements. For example, a method declaration is represented by a class named `JCMethodDecl`, a wildcard by a class named `JCWildcard`. We map the AST nodes into nodes in our graph model and we use these class names to identify the nodes. We store the names in a property called `nodeType` (an enumeration containing `JCMethodDecl` etc.) attached to each node.

Some AST node have fields pointing to other AST nodes to represent parent-child relationships. For example, a binary expression is represented by a `JCBinary` class, which contains a field named `lhs` pointing to its left child expression and another field named `rhs` pointing to its right child expression. We map these names to labelled edges. As an example, in our model `JCBinary` is mapped to a node with a property (`nodeType:"JCBinary"`), two edges labelled respectively `LHS` and `RHS` connecting two separate nodes with a property (`nodeType:"JCExpression"`).

The complete list of relationships consists of around 120 different labelled edges and can be accessed on the project home page [11]. It includes `IF_THEN` to represent the relationship of an if statement and its first branch, `IF_ELSE` to represent the relationship of an if statement with its last branch, `STATEMENTS` to represent a block enclosing a list of statements etc. Note that we also add a relation `ENCLOSES` on top of these (i.e. an overlay) to represent the general relationship of a node enclosing another one.

This model lets us retrieve source-code elements using the `nodeType` property as a search key. Searching for certain source-code patterns becomes a graph traversal problem with the correct sequence of labelled edges and nodes. For example, finding a while loop within a specific method can be seen as finding a node `JCMethodDecl` connected to a `JCWhileLoop` with a directed labelled edge `ENCLOSES`.

Additionally, we define two properties on the nodes that are useful for software engineering metrics queries: `size` (e.g. number of lines of code or character count that this node represent in the source file) and `position` (e.g. starting line number and/or offset of this node in the source file). These are implemented easily as the *javac* front-end provides them for every syntactic object.

4.2 Overlays

The most obvious graph model of source code is the abstract syntax tree. However, even finding the source-language type of a node representing a variable use is difficult as we need an iterative search outwards through scopes to find its *binding* occurrence in a definition which is where the type is specified.

What we need is an additional edge between a variable use and its binding occurrence—this additional relation is an *overlay*. For example, we connect method and constructor invocations to their declarations with a labelled edge `HAS_DECLARATION` and variables and field uses to their respective definitions with a labelled edge `HAS_DEFINITION` to access these directly. Similarly we might want to precompute a program's call graph and store it as an overlay relation instead of searching for call nodes on demand.

Sometimes it is also desirable to distinguish different instances of a same labelled edge. For example, while a class declaration contains a list of definitions, it is useful to distinguish between field method definitions to facilitate querying. This is why we distinguish these with `DECLARES_FIELD` and `DECLARES_METHOD` overlays.

Similarly, it is difficult to know ahead of time all possible information that users may want to query. For example, users may want to query for overloaded methods. Such a query could be defined by processing all methods nodes available in our model and grouping them by fully qualified name and returning the groups that occur more than once. However, this query would be much more efficient if method nodes had a pre-computed (overlay) property indicating whether or not they are overloaded methods.

As explained earlier, the graph data model relies on semi-structured data so new overlays can be added without the schema to change. As a result, our model can easily cache results of queries as additional (overlay) relations or nodes.

We now describe the pre-computed overlays in our model corresponding to the program views described in Section 3.

Type Hierarchy

We overlay the type hierarchy defined in the program on top of the AST. We connect nodes that represent a type declaration with a directed edge to show a type relation. These edges are labelled to distinguish between an `IS_SUBTYPE_IMPLEMENTS` or `IS_SUBTYPE_EXTENDS` relation. These overlays let us map the problem of finding the subtypes of a given type as a graph path-finding query. As a result, we do not need to examine class and interface declarations to extract the information. Note that we prefer to add overlays for one-step relations (e.g. direct subtype) as transitive closure can be simply expressed and computed by a graph query language (see Section 5).

Override Hierarchy

Similarly, we overlay the override hierarchy of methods defined in the program on top of the AST. We connect method declaration nodes in the AST with the parent method they override using labelled edges `OVERRIDES`. As a result, we can query the override hierarchy directly instead of examin-

ing each class and method declaration to extract the information.

Type Attribution

We overlay a property called `actualType` on each expression node in the AST. It holds the string representation of the resolved type of an expression. This facilitates common source-code queries such as finding a method call on a receiver of a specific type. Indeed, this query would simply retrieve the method invocation nodes (labelled `JCMethodInvocation`) and check that the property `actualType` is equal to the desired type. Similarly, one can find covariant array assignments by comparing the `actualType` property of the left-hand and right-hand side of an assignment node (labelled `JCAssign`).

The Java compiler also defines an enumeration of different type kinds that an expression can have. These include `ARRAY`, `DECLARED` (a class or interface type), `INT`, `LONG`. We store these values in a `typeKind` property.

Method Call Graph

We construct the method call graph by extracting the method invocations from the body of each method declaration. Each method invocation is resolved and associated with its method declaration based on two filters: the static type of the receiver (weak call graph) and all possible subtypes of the receiver static type (rich call graph). We connect a method invocation node with its resolved method declaration(s) with a labelled edge `CALLS` for the weak call graph and `CALLS_DYN` for the rich call graph.

Typical source-code queries such as finding all the methods called within the entry point of a class can be seen as a traversal query on the generated call graph. In addition, we can easily find recursive methods by finding method declarations nodes with a `CALLS` loop.

Data Flow

Dataflow analysis is a a vast topic; this work describes only a simple set of data flow queries. We connect each expression with the declaration of the variables that they use using a `GEN` edge. Similarly, variables that are necessarily written to are connected to the expression with a `KILLS` edge. Such information is useful for performing certain data flow analysis such as live variable analysis; the definitions above are conservative for this—we consider the conservative set of variables that *must* be written to, but only these which *may* be read.

Next, we construct the read and write dependency graphs for each block. We inspect each block declarations (e.g method declarations, loops) to extract any field and variable accesses. We connect the blocks to the field or variable declarations that they read from or write to using `READS` and `WRITES` edges. The `READS` set of a block can be seen as the union of all the outgoing nodes connected with a `GEN` edge from all the expressions contained inside that block. Similarly, the `WRITES` set can be seen as the union of all outgoing nodes connected with a `KILLS` edge from all the expressions contained inside that block. This additional overlay is useful to easily compose program comprehension queries such as finding out which fields are written to by a given method declaration.

5. Source Code Queries via Graph Search

In this section, we describe how one can write source-code queries using our model. We show that a query language specialised for graph traversal that supports *graph pattern matching*, *path finding*, *filtering* and *aggregate functions* can express all queries expressible in existing Datalog approaches.

Hajiyev et al. highlighted that a source-code query language needs to support recursive queries to achieve high expressiveness [20]. Indeed, certain queries such as traversing the subtype hierarchy or walking tree structures require recursion. To this end, they suggested Datalog, which supports built-in transitive closure. *Graph pattern matching* (i.e the ability to match certain parts of a graph in a variable) can be seen as binding a set of nodes in a variable based on the values of node properties and labelled edges. *Path finding* operations have also built-in transitive closure in order to traverse the full graph.

We show examples of source-code queries using *CYPHER*, a graph query language for the Neo4j database [9]. Its use obviates the need to for complex translation to other database query languages such as Codequest [20]. CYPHER also supports aggregate functions such as `COUNT()`, `SUM()`, which appear only as extensions to Datalog. Such operations are useful to express software metric queries because they often need to apply arithmetic operations on a group of data.

The next subsections describe in detail several of source-code queries written in CYPHER.

5.1 Implementing java.lang.Comparator

We introduce the syntax of CYPHER by showing a simple query to retrieve all classes implementing the interface `java.util.Comparator` in Figure 1 (Listing 1).

The `START` clause indicates a starting point in the graph. We then iteratively bind in the variable `p` to all the nodes that have a property `nodeType` equal to `ClassType` by looking up the index of nodes called `node_auto_index`. Next, we use the `MATCH` clause to generate a collection of subgraphs matching the given graph pattern. Specifically, we look for subgraphs starting at a node bound to a variable `n` which has an outgoing edge labelled `IS_SUBTYPE_IMPLEMENTS` connected to `p`. Next, we eliminate any subgraphs where `p` is not of type `java.util.Comparator` by using a `WHERE` clause on the property `fullyQualifiedName`. Finally, we return the elements of all the matched subgraph as a list of tuples.

Listing 1

```
START p=node:node_auto_index(nodeType='ClassType')
MATCH n-[:IS_SUBTYPE_IMPLEMENTS]->p
WHERE p.fullyQualifiedName='java.util.Comparator'
RETURN n,p;
```

Listing 2

```
START m=node:node_auto_index(nodeType='ClassType')
MATCH path=n-[r:IS_SUBTYPE_EXTENDS*]->m
WHERE m.fullyQualifiedName='java.lang.Exception'
RETURN path;
```

Listing 3

```
START m=node:node_auto_index(nodeType='JCMethodDecl')
MATCH c-[:DECLARES_METHOD]->m-[:CALLS]->m
RETURN c,m;
```

Listing 4

```
START n=node:node_auto_index(nodeType ='JCWildcard')
RETURN n.typeBoundKind , count(*)
```

Listing 5

```
START n=node:node_auto_index(nodeType='JCAssign')
MATCH lhs<-[:ASSIGNMENT_LHS]-n-[:ASSIGNMENT_RHS]->rhs
WHERE
    has(lhs.typeKind) AND lhs.typeKind='ARRAY' AND
    has(rhs.typeKind) AND rhs.typeKind='ARRAY' AND
    lhs.actualType <> rhs.actualType
RETURN n;
```

Listing 6

```
START m=node:node_auto_index(nodeType ='JCMethodDecl')
MATCH c-[:DECLARES_METHOD]->m
WHERE m.name <> '<init>'
WITH
    c.fullyQualifiedName+":"+m.name as fullyQualifiedMethod,
    count(*) as overloadedCount
WHERE overloadedCount > 1
RETURN fullyQualifiedMethod, overloadedCount
ORDER BY overloadedCount DESC;
```

Figure 1. Examples of source-code queries written in CYPHER

Start node (n)	End node (p)
{nodeType:"JCClassDecl", lineNumber:36, position:1654,size:473, simpleName:"StringComparator", fullyQualifiedName:"org.hsqldb.lib.StringComparator"}	{nodeType:"ClassType", fullyQualifiedName:"java.util.Comparator"}
{nodeType:"JCClassDecl", lineNumber:56, position:2444,size:9564, simpleName:"SubQuery", fullyQualifiedName:"org.hsqldb.SubQuery"}	{nodeType:"ClassType", fullyQualifiedName:"java.util.Comparator"}
{nodeType:"JCClassDecl", lineNumber:86, position:3576,size:417, simpleName:"", fullyQualifiedName:"<anonymous java.util.Comparator>"}	{nodeType:"ClassType", fullyQualifiedName:"java.util.Comparator"}

Figure 2. Example output for the query *implementing java.lang.Comparator* on the program *Hsqldb*

Figure 2 shows an extract of the output from executing this query on the source code of the program *Hsqldb*.

5.2 Extending java.lang.Exception

We can modify this query to find all classes that are directly or indirectly subtypes of `java.lang.Exception` as shown in Figure 1 (Listing 2). We generate a collection of all subgraphs starting at a node bound to the variable `n` that is connected to a node `m` that represents `java.lang.Exception`. We use the asterisk (`*`) to specify a form of transitive closure: here that the path to reach `m` can be of arbitrary length as long as it is solely composed of edges of type `IS_SUBTYPE_EXTENDS`. Finally, we bind all the subgraphs that were matched into a variable `path`, which we return.

Figure 3 shows an extract of the output from executing this query on Hsqldb. The class `FileCanonicalizationException` extends `BaseException`, which itself extends `java.lang.Exception`

5.3 Recursive Methods with their Parent Classes

In Figure 1 (Listing 3) we show how to search for classes containing recursive methods. First, we look up the index of nodes to find all method nodes that have the property `nodeType` equal to `JCMethodDecl` and iteratively bind

Node (n)	Outgoing relationship (r)
{nodeType:"JCClassDecl", lineNumber:2002, position:83668,size:1388, simpleName:"FileCanonicalizationException", fullyQualifiedName:"org.hsqldb.persist.LockFile.FileCanonicalizationException"}	:IS_SUBTYPE_EXTENDS
{nodeType:"JCClassDecl", lineNumber:1937, position:81667,size:1684, simpleName:"BaseException", fullyQualifiedName:"org.hsqldb.persist.LockFile.BaseException"}	:IS_SUBTYPE_EXTENDS
{nodeType:"ClassType",fullyQualifiedName:"java.lang.Exception"}	

Figure 3. Example output for the query *directly or indirectly extending java.lang.Exception* on the program *Hsqldb*

them to the variable `m`. Next we specify a graph pattern using the `MATCH` clause that generates a collection of all subgraphs starting at a node `c` (a class declaration) that is connected to `m`. In addition, we specify that the method `m` is calling itself by specifying an edge `CALLS` from `m` to `m` (weak call graph).

5.4 Wildcards

Parnin et al. investigated the use of generics by undertaking a corpus analysis of Java code [23]. In this example, we show how a query can investigate the use of generic wildcards. We would like to group the different kinds of wildcards by the number of their occurrences which appear in the source code. CYPHER provides aggregate functions such as `count(*)` to count the number of matches to a grouping key. We can use this function in a `RETURN` clause to count the number of occurrences based on typeBoundKind property of nodes of type `JCWildcard` as shown in Figure 1 (Listing 4). Figure 4 shows a possible output of running this query.

n.typeBoundKind	count(*)
UNBOUNDED_WILDCARD	67
EXTENDS_WILDCARD	32

Figure 4. Result for grouping Java wildcards by kinds

5.5 Covariant Array Assignments

We recently conducted a corpus analysis to investigate the use of covariant arrays and found that these are rarely used in practice [30]. In Figure 1 (Listing 5), we show how one could find uses of covariant arrays in assignment context.

First, we look up nodes of type `JCAssign`, which represent an assignment in the AST. We iteratively bind them to the variable `n`. Next, we use a `MATCH` clause to decompose an assignment into its left and right children, which we bind respectively in the `lhs` and `rhs` variable. We ensure that these nodes are both tagged as holding an expression of type `ARRAY` by restricting the `typeKind` property using a `WHERE` clause. Finally, we also ensure that the resolved type (e.g. `Object[]`) and `String[]` of the left hand side and right hand side of the assignment are different by checking the `actualType` property, which holds the string representation of the type of expression.

5.6 Overloaded Methods

Gil et al. investigated how Java programmers make use of overloading by undertaking a corpus analysis of Java code [19]. This research question can be partially answered by composing a query which retrieves overloaded methods. Figure 1 (Listing 6) shows how to generate a report of which method names in which classes are overloaded along with the degree of overloading.

First, we bind all nodes of type `JCMethodDecl` to variable `v`. We then generate all subgraphs connecting `m` with a node `c` (a class declaration) with a `DECLARES_METHOD` relation. We add an additional constraint using a WHERE clause to eliminate subgraphs where the method nodes are constructors (methods named `<init>`). Next, we generate the fully qualified name of the method by concatenating the fully qualified class declaration name with the method name. We use this as the grouping key for the `count(*)` aggregate function. Finally, we return the fully qualified method names that appear more than once (i.e overloaded methods) together with how many times they occur.

6. Implementation

In this section, we give an overview of the implementation of our source-code querying system.

Our current prototype consists of a Java compiler plug-in to analyse source-code files and a Neo4j graph database to store the source-code information converted in our graph model. We selected Neo4j [9] as our database back end because it is established in the community, has a Java API and also provides a graph query language that supports the necessary features described in Section 5. The source code of our prototype (1300 lines of Java code) can be accessed on the project homepage [11].

The first part of our system is responsible for parsing Java source-code files, analysing them and converting them into our graph model. To this end, we developed a Java compiler plug-in that accesses the Java compiler structures and functionalities via API calls [29]. We used this mechanism be-

cause a compiler plug-in interacts in harmony with existing building tools such as *ant* and *maven* that rely on the Java compiler. Users of our tool can therefore compile their programs normally without running external tools. However, the Java compiler requires the entire project to compile before it can be processed. There are alternatives such as Eclipse JDT [2] that do not have this requirement but that do not integrate so smoothly with standard building tools.

Our compiler plug-in traverses the Abstract Syntax Tree (AST) of each source-code file. It translates the AST elements and their properties to graph nodes in Neo4j. Links between AST elements are translated to labelled edges between the created nodes. We retain the full AST and do not throw away any information. Additionally, we implemented the type hierarchy, type attribution and call graph overlays described in Section 4.

Neo4j provides indexing facilities that map a key to a set of records. We use these facilities to index nodes based on the `nodeType` property in order to improve the speed of specific AST-node-retrieval queries. Finally, source-code queries can be executed by writing CYPHER queries through the shell interface provided with the Neo4j server.

7. Experiments

In this section, we evaluate the scalability and performance of our system. We describe a set of experiments and their results demonstrate that our system scales well.

First, we evaluate the performance of queries related to code exploration. Next, we investigate the performance of queries related to language design research that make use of richer source-code information. Finally, we also compare performance of certain queries supported by JQuery [21].

7.1 Setup

We intercept all calls to *javac* by replacing its binary on the standard search path with an executable file that calls *javac* with our compiler plug-in. As a result, any Java build tools (such as *ant* or *maven*) that use *javac* to compile files will automatically process the source code in the graph database.

We selected four Java open source programs of various sizes. We summarise them in Figure 5. In addition, we created a corpus of 12 Java projects totalling 2.04 million lines of code, which can be accessed on the project home page. It includes *Apache Ant*, *Apache Ivy*, *AoI*, *HsqlDB*, *jEdit*, *Jspwiki*, *Junit*, *Lucene*, *POI*, *Voldemort*, *Vuze* and *Weka*.

We ran the experiments on two machines with Neo4j standalone server 1.9.M04 and OpenJDK 1.7.0:

- A typical personal computer used for programming tasks: a MacBook Pro with 2.7GHz Intel Core i7, 8GB of memory and running OSX 10.8.2
- An Amazon EC2 instance (m3.2xlarge) with 30GB and EBS storage running Linux. We provide the virtual image on the project homepage for reproducibility.

We ran the database server with default settings. However, it is possible to tune certain configuration parameters such as memory usage to gain additional performance.

Our experiments consists of running different queries on the programs and report the response time. We distinguish between a *cold* system and a *warm* system to investigate the impact of caching on performance.

- We report the response time of a query after restarting the querying system each time (*cold*).
- We run the same queries twice and report the faster response time (*warm*).

In a real scenario, it is possible for a developer to run the same query several times. In practice, a production source querying system could even warm the cache ahead of time with common queries to enhance performance. Queries having similar sub-queries in common will also benefit from the cache.

Application	Description	# of Java files	LoC
Junit 4.2	Java unit test framework	79	3206
Apache Ivy 4.2.5	Dependency manager	601	67989
Hsqldb 2.2.8	Relational database engine	520	160250
Vuze	Bitorrent client	3327	509698
Corpus	12 open source software	13559	2038832

Figure 5. Summary of Java projects used for the experiments

7.2 Queries

Code Exploration Queries

Source code querying systems are useful to developers in order to rapidly explore a codebase by retrieving some elements. We ran the following queries that explore a given codebase (the syntax for them is described in Section 5):

1. Find all classes that directly implement `java.util.Comparator` (Figure 1, Listing 1)
2. Find all classes that directly or indirectly extend `java.lang.Exception` (Figure 1, Listing 2)
3. Find all recursive methods *together with* their parent classes (Figure 1, Listing 3)

Research Queries

Source code querying systems can also be used by programming language researchers to retrieve usage information about certain features or patterns. For each project, we ran the following research queries as these were investigated in previous work (the syntax for them is described in Section 5):

1. Report a ratio of the usage of generic wildcards (Figure 1, Listing 4)
2. Find all covariant array assignments (Figure 1, Listing 5)

3. Find all overloaded methods (Figure 1, Listing 6)

7.3 Results

We show the absolute response time to the queries in Figure 7. Queries 4 and 6 have a response time less than half a tenth of a second. This shows that while our system can efficiently traverse graph structures, aggregate operations are also efficiently executed. Figure 6 presents the execution time expressed as ratios relative to the program *Hsqldb*. The results shows that the query response times scale linearly with increasing program size.

We found that running queries with a hot cache decreases response time by a factor of 3x to 5x compared to a cold cache. Finally, we also found that a larger machine (EC2 versus MacBook even on millions of lines of code) does not directly improve performance of queries. We note a net improvement for the largest programs (Corpus) for Queries 3 and 5 with a cold cache.

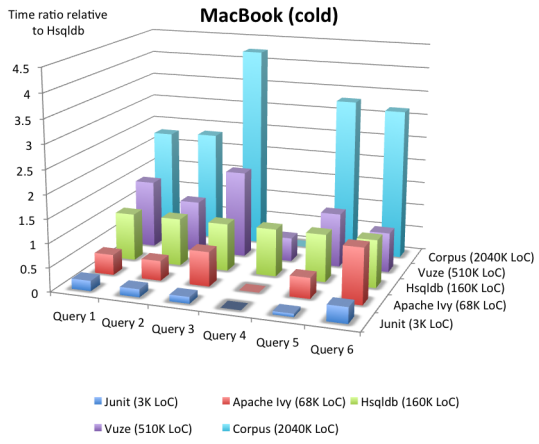

Figure 6. Queries times on MacBook relative to Hsqldb (cold)

7.4 Comparison with Other Systems

Unfortunately, we could not find a working implementation for recent system such as Codequest and JTL. However, the authors reported absolute times of general source-code queries executed on their system, which we briefly compare with. The reader should note that it is difficult to make a fair comparison since the queries were executed on slower machines than what we used for our experiments. However, our system stores more information about the source code of programs, which also impacts performance. For example, CodeQuest, JTL and JQuery do not support method bodies, generics or types of expressions.

JTL The authors reported execution times between *10s* and *18s* for code exploration queries carried out on *12,000* random classes taken from the Java standard library. However, it is unclear whether the experiment were performed with a cold or warm system.

```
// 1. recursive methods
method(?C,?M), calls(?M,?M,?L);

// 2. classes declaring public fields
class(?C), field(?C,?F), modifier(?F, public);

// 3. overloaded methods
method(?C1, ?M1), method(?C2, ?M2),
likeThis(?M1,?M2)
```

Figure 8. Test queries for JQuery

We reported times of 0s to 12s running on a *cold* system and 0s to 3s with a *warm* system by running queries on a larger data set (a Corpus of over *13,000* Java source files and 2.04 million lines of code). Note that our setup is given an advantage because the JTL experiments were reported on a 3GHz Pentium 4 processor with 3GB of memory whereas we used a more modern machine (MacBook Pro with 2.7GHz Intel Core i7 with 8GB of memory).

Codequest The authors ran various queries on different database systems. We compare with the fastest times they report, which were found on a quad Xeon 3.2GHz CPU with 4GB of memory and using XSB (a state of the art deductive database). The authors reported times between *0.1s* and *100s*. The slowest query is looking for methods overriding a parent method defined as `abstract`. In the Codequest system, the overriding relation between methods is computed on the fly. In our system, the override relation is already pre-computed as an *overlay* on the AST.

JQuery We created three queries in JQuery: finding recursive methods, classes declaring public fields and overloaded methods as shown in Figure 8. We tested them on the *ivy* program (68kLoC) because JQuery failed to index larger projects such as *Hsqldb*. This deficiency was already reported by other work [15, 20]. All queries executed in a couple of seconds except that for overloaded methods which took over 30s on our MacBook machine.

8. Related work

In this section, we review existing source-code querying systems and highlight what information about the program is lost and the consequential restrictions on source-code queries.

Source code querying systems can be classified in two categories:

1. **software**: query the source code of one piece of software in particular.
2. **repository**: query the source code of multiple projects at the same time.

We provide a detailed comparison of software-level source-code querying languages in a previous paper [28]. We evaluated whether their expressiveness suffices for language

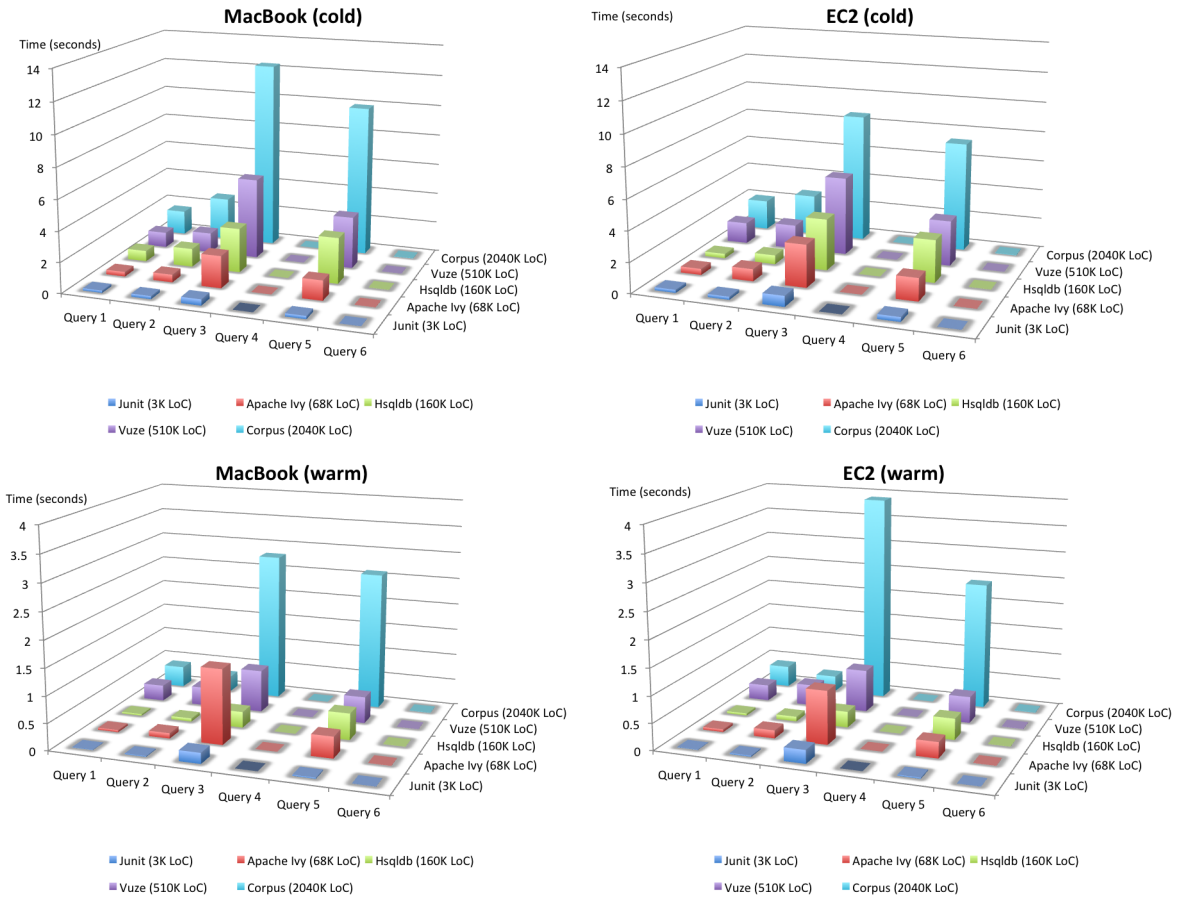

Figure 7. Absolute queries times on MacBook and EC2. (Notice the varying y-axis scale)

design research. To this end, we studied several use cases of recent Java features and their design issues—investigating their expressibility as queries in the source-code querying languages we examined. For example, one use case queries for covariant array assignments. We found that only two query languages (Soul and .QL) listed below provide the minimal features required to express all our use cases.

8.1 Software Querying Systems

Java Tools Language (JTL) is a logic-paradigm query language to select Java elements in a code base and compose data-flow queries [15]. The implementation analyses Java bytecode classes. Since JTL is based on bytecode analysis rather than source-code analysis, several queries are inherently restricted. For example, Java generics are compiled to casts during the bytecode translation.

Browse-By-Query (BBQ) reads Java bytecode files and creates a database representing only classes, method calls, fields, field references, string constants and string constant references [1]. This database can then be interrogated through English-like queries.

SOUL is a logic-paradigm query language [17]. It contains an extensive predicate library called *CAVA* that matches queries against AST nodes of a Java program generated by the *Eclipse JDT*.

JQuery is a logic-paradigm query language built on top of the logic programming language *TyRuBa* [21]. It analyses the AST of a Java program by making calls to the Eclipse JDT. JQuery includes a library of predicates that lets the user query Java elements and their relationships.

Codequest is a source-code querying tool combining Datalog as a query language and a relational database to store source-code facts [20]. Datalog queries are translated to optimised SQL statements. The authors report that this approach scales better compared to JQuery. However, only a subset of Java constructs are supported and types of expressions are not available. For example, one could not locate assignments to local variables of a certain type.

.QL is an object-oriented query language. It enables programmers to query Java source code with queries that resemble SQL [16]. This design choice is motivated as reducing the learning curve for developers. In addition, the authors

argue that object-orientation provides the structure necessary for building reusable queries. A commercial implementation is available, called *SemmleCode*, which includes an editor and various code transformation.

Jackpot is a module for the *NetBeans IDE* for querying and transforming Java source files [5]. Jackpot lets the user query the AST of a Java program by means of a template representing the source code to match. However, Jackpot does not provide any type information for AST nodes

PMD is a Java source-code analyser that identifies bugs or potential anomalies including dead code, duplicated code or overcomplicated expressions [10]. One can also compose custom rules via an *XPath* expression that queries the AST of the program. However, types of expressions are not available.

8.2 Repository Querying Systems

Sourcerer is an infrastructure for performing source-code analysis on open source projects [13]. It consists of a repository of Java projects downloaded from Apache, Java.net, Google Code and Sourceforge. Source code facts of the projects in the repository are extracted and stored in a relational database that can be queried. Information available include program *entities* such as classes or local variable declarations. It also includes *relations* between entities such as `CONTAINS` (e.g. a class containing a method) or `OVERRIDES` (e.g. a method overrides a method).

Sourcerer focuses on storing structural information; we consider this as an overlay to the AST, but in sourcerer the AST is then discarded meaning that queries involving statements within a method or types of expressions are not possible.

Boa is an infrastructure for performing metadata mining on open source software repositories [18]. It indexes metadata information about projects, authors, revision history and files. However, it stores no source-code facts. The infrastructure provides a query language to mine information about the repositories. For example, one can query the repositories to find out the ten most used programming languages. Currently Boa indexes more than a million projects from Github, Google Code and Sourceforge. Overall scalability is achieved by using a Hadoop as a map-reduce framework to process the data in parallel.

9. Discussion and Further Work

In this section we discuss further work as well as an application of our work that aims to facilitate corpus analysis.

9.1 Specifying and Implementing Overlays

At the moment, our system stores as overlays those relations corresponding to compiler data structures, such as control-flow graph successors and predecessors, call graph and the like *as the authors felt useful*. However, this is rather ad-hoc and we now turn to a more disciplined approach. Ideally, we would like overlays to be seen as cached results of queries—thus we *specify* the binding occurrence of a variable use as the result of a search; this can be *implemented* either in terms of an additional pre-computation for every variable at start-up time, or in terms of a cached search result.

This exposes an expressiveness question: suppose we wish to provide overlay information for data-flow purposes (e.g. which assignments *reach* a given variable use, or which variables may alias at a given variable use). In general such data-flow analysis involves fixed-point computations; these are in general rather more complex than simple transitive closure (e.g. Andersen's simple points-to analysis is equivalent to *dynamic* transitive closure). We plan to investigate whether the query language can be refined to permit direct representation of data-flow computations along the lines of [26].

This would be very attractive, in that a user query may then consist of a number of definitions followed by one or more queries as before. The definitions can then be implemented not as simple function calls during search (which could be slow if they are used many times), but as pre-calculating and storing overlay relations before the query(ies) themselves are evaluated.

9.2 A Corpus Querying Platform

Empirical studies are useful to understand how programming language features or idioms are used in practice. They provide answers to questions and hypotheses that can influence the evolution of programming languages. There are several proposals such as Qualitas Corpus (a corpus of open source software) that aim to help language researchers examine code [27]. However, conducting studies can be difficult, time consuming and difficult to replicate by the research community. For example, analysing the usage frequency of a specific idiom requires complex static analysis and reporting tools. Source-code querying infrastructures can ease the process: researchers can express a research question as a source-code query over a corpus of software. However, as discussed above, current solutions make ad-hoc tradeoffs between scalability, amount of source-code detail held in the database, and expressiveness of queries.

We are therefore developing an infrastructure based on our work, called *Wiggle*, which tackles these problems. Researchers can express programming language research questions acting over a corpus of Java software and get a generated report to their queries. Currently, queries are expressed via CYPHER. However, certain queries such as finding covariant array assignments can be difficult to write. Hence, we are also designing a user-friendly source-code query language which compiles to CYPHER. It inspired from query-by-example [32] and SQL forms in order to make it intuitive to use. For example, one can query for covariant array assignments by first composing a *structural query* which finds assignments between two array expressions and bind their

element types into two variables S and T as follows: `T[] = S[]`. Next, we filter the set of results where the right hand-side element type is a subtype of the left hand-side using *semantic constraints*: `S <: T`.

We also plan to extend our infrastructure with support for version history [25]. This would enable researchers to ask a wider range of questions on corpus and language evolution—such as adoption of certain development practices, new libraries or language features over time.

9.3 Query Optimisation and Cost

On large source-code corpora it is important to restrict queries to near-linear time, $O(n)$. This is more subtle than first appears, as while corpora may contain many classes and methods ($O(n)$), each method is itself effectively bounded in size and hence $O(1)$. Thus while the query: "find all method names p and method names q" is likely to be quadratic in n because of the effective join, the query: "find all method names p and method names q such that p calls q" is likely to be linear. There are two issues: one is that we wish to optimise queries like the above so that they are calculated in linear time whenever possible (and perhaps to warn if this is not possible). The other concerns more program-wide queries which are not in general linear in n, such as points-to analysis. An approach here is to use a less-accurate algorithm initially (e.g. Stensgard's algorithm which is almost linear) and then to refine its results on demand.

10. Conclusion

This paper presented the design and implementation of a source-code querying system using a *graph database* that stores full source-code detail and scales to program with millions of lines of code.

First, we described a model for Java source code based on the *graph data model*. We introduced the concept of *overlays*, allowing queries to be posed at a mixture of syntax-tree, type, control-flow-graph and dataflow levels (Section 4).

Next, we showed that a graph query language can naturally express queries over source code by showing examples of code exploration and language research queries written in CYPHER (Section 5).

Finally, we evaluated the performance of our systems over programs of various size (Section 7). Our results confirmed that graph databases provide an efficient implementation for querying programs with millions of lines of code while also storing full source-code detail.

Acknowledgments

We thank Alex Buckley, Joel Borggrén-Franck, Jonathan Gibbons and Steffen Lösch for helpful comments. This work was supported by a Qualcomm PhD studentship.

References

[1] BBQ. `http://browsebyquery.sourceforge.net/`.

[2] Eclipse Java development tools. `http://www.eclipse.org/jdt`.

[3] Findbugs. `http://findbugs.sourceforge.net`.

[4] Google code search. `http://code.google.com/codesearch`.

[5] Jackpot. `http://wiki.netbeans.org/Jackpot`.

[6] Java 8: support for more aggressive type-inference. `http://mail.openjdk.java.net/pipermail/lambda-dev/2012-August/005357.html`, .

[7] Java coding standard. `http://lars-lab.jpl.nasa.gov/JPL_Coding_Standard_Java.pdf`, .

[8] Object Query Language. `http://www.ibm.com/developerworks/java/library/j-ldn1`, .

[9] Neo4j Graph Database. `http://www.neo4j.org/`.

[10] PMD. `http://pmd.sourceforge.net/`.

[11] Wiggle Project. `http://www.urma.com/wiggle`.

[12] R. Angles and C. Gutierrez. Survey of graph database models. *Computing Surveys*, 40(1):1, 2008.

[13] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An infrastructure for the large-scale collection and analysis of open-source code. *Science of Computer Programming*, 2012.

[14] S. Breu and T. Zimmermann. Mining aspects from version history. In S. Uchitel and S. Easterbrook, editors, *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*. ACM Press, September 2006.

[15] T. Cohen, J. Y. Gil, and I. Maman. JTL: The Java tools language. In *OOPSLA*, 2006.

[16] O. de Moor, M. Verbaere, and E. Hajiyev. Keynote address: .QL for source code analysis. In *SCAM*, 2007.

[17] C. De Roover, C. Noguera, A. Kellens, and V. Jonckers. The SOUL tool suite for querying programs in symbiosis with Eclipse. In *PPPJ*, 2011.

[18] R. Dyer, H. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *ICSE*, 2013.

[19] J. Gil and K. Lenz. The use of overloading in Java programs. In *ECOOP*, 2010.

[20] E. Hajiyev, M. Verbaere, and O. De Moor. Codequest: Scalable source code queries with datalog. *ECOOP 2006–Object-Oriented Programming*, pages 2–27, 2006.

[21] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *AOSD*, 2003.

[22] A. Kellens, C. De Roover, C. Noguera, R. Stevens, and V. Jonckers. Reasoning over the evolution of source code using quantified regular path expressions. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 389–393. IEEE, 2011.

[23] C. Parnin, C. Bird, and E. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *Mining Software Repositories*, 2011.

[24] S. Paul and A. Prakash. A query algebra for program databases. *Software Engineering, IEEE Transactions on*, 22 (3):202–217, 1996.

[25] R. Stevens. Source Code Archeology using Logic Program Queries across Version Repositories. Master's thesis, Vrije Universiteit Brussel, 2011.

[26] A. Stone, M. Strout, and S. Behere. May/must analysis and the dfagen data-flow analysis generator. *Information and Software Technology*, 51(10):1440–1453, 2009.

[27] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The qualitas corpus: A curated collection of java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336–345. IEEE, 2010.

[28] R. Urma and A. Mycroft. Programming language evolution via source code query languages. In *Proceedings of the ACM 4th annual workshop on Evaluation and usability of programming languages and tools*, pages 35–38. ACM, 2012.

[29] R.-G. Urma and J. Gibbons. Java Compiler Plug-ins in Java 8. *Oracle Java Magazine*, 2013.

[30] R.-G. Urma and J. Voigt. Using the OpenJDK to investigate covariance in Java. *Oracle Java Magazine*, 2012.

[31] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the 48th annual Southeast regional conference*, page 42. ACM, 2010.

[32] M. M. Zloof. Query by example. In *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pages 431–438. ACM, 1975.