Object Graph Programming

Aditya Thimmaiah
The University of Texas at Austin
Austin, Texas, USA
auditt@utexas.edu

Leonidas Lampropoulos
University of Maryland
College Park, Maryland, USA
leonidas@umd.edu

Christopher J. Rossbach
The University of Texas at Austin
Austin, Texas, USA
rossbach@cs.utexas.edu

Milos Gligoric
The University of Texas at Austin
Austin, Texas, USA
gligoric@utexas.edu

ABSTRACT

We introduce Object Graph Programming (OGO), which enables reading and modifying an object graph (i.e., the entire state of the object heap) via declarative queries. OGO models the objects and their relations in the heap as an object graph thereby treating the heap as a graph database: each node in the graph is an object (e.g., an instance of a class or an instance of a metadata class) and each edge is a relation between objects (e.g., a field of one object references another object). We leverage Cypher, the most popular query language for graph databases, as OGO's query language. Unlike LINQ, which uses collections (e.g., List) as a source of data, OGO views the entire object graph as a single "collection". OGO is ideal for querying collections (just like LINQ), introspecting the runtime system state (e.g., finding all instances of a given class or accessing fields via reflection), and writing assertions that have access to the entire program state. We prototyped OGO for Java in two ways: (a) by translating an object graph into a Neo4j database on which we run Cypher queries, and (b) by implementing our own in-memory graph query engine that directly queries the object heap. We used OGO to rewrite hundreds of statements in large open-source projects into OGO queries. We report our experience and performance of our prototypes.

KEYWORDS

Object graph, graph database, query, Cypher

ACM Reference Format:
Aditya Thimmaiah, Leonidas Lampropoulos, Christopher J. Rossbach, and Milos Gligoric. 2024. Object Graph Programming. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623319

1 INTRODUCTION

Declarative programming [53], focusing on the *what* rather than the *how*, has grown into the predominant way of programming in an increasing number of domains. For instance, Structured Query Language (SQL), a canonical example of the declarative paradigm, is the primary query language for most relational database management systems [9, 28, 69, 70]. At the same time, NoSQL databases have been gaining traction. In particular, the space of *graph databases* [4, 50, 91, 92] is growing at a rapid pace, as they have been shown to be a great fit for tasks such as fraud detection, drug discovery [100], recommendation engines, and data visualization [24, 58]. Graph databases store data as property graphs [3, 44, 64], which emphasize relationships between data.

A *property graph* (graph for short) contains nodes $\mathcal{N}$ and edges $\mathcal{R}$ denoting relationships between nodes. Each node is assigned a *label* $\mathcal{L}$ and contains an arbitrary set of *properties* $\mathcal{P}$: mappings from nodes to values. Edges also have a label (sometimes called type in the literature) and an arbitrary set of properties. Querying, updating, and administering of such a graph is performed with a *graph query language*. Cypher [35, 45], initially developed as part of the Neo4j project [73], is currently the most popular graph query language [46, 98]. Cypher is a declarative language, in many ways similar to SQL, which emphasizes simplicity and expressivity. As an example, to get the values of all nodes in a graph database we could run the following query: `match(n : Node) return n.val`. Although graph databases have been used for various tasks, *the power of property graphs and graph query languages has yet to be used to enhance developers' experience.*

Our key insight is that an object graph [39], i.e., in-memory program state available at the execution time, can be seen as a property graph. We believe that being able to query object graphs during development, testing, and debugging will substantially extend the power of programming languages and tools.

We present *Object-Graph Programming* (OGO) that enables querying and updating an object graph via declarative queries. OGO treats a given object graph as a graph database: each node in the graph is an object (e.g., an instance of a class or an instance of a meta class) and each edge is a relation between objects (e.g., a field of one object references another object). We leverage Cypher as OGO's query language. This gives rise to endless opportunities to leverage OGO for programming, analyses, and tool development. We describe several potential use cases where OGO can be applied.

OGO provides a powerful and expressive way for writing assertions and program invariants [71, 74]. Assertions written using OGO not only can access the local program state, but they can also check any aspect of the dynamic state of a program.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3623319

For example, we could assert that there is never more than one instance of a specific (singleton) class. Moreover, like LINQ, OGO can be used for querying collections of data and even implementing these collections. Unlike LINQ, OGO, at the moment, requires developers to cast their results to appropriate type, as we do not guarantee type safety, which is similar to working with the `java.sql` [85] package. At the same time, OGO can query any collection (e.g., an n dimensional array) without requiring a user to implement any specific interface.

In this paper, we show the power of OGO with these two use cases, but we envision many further benefits and potential uses.

For example, OGO could facilitate dynamic program analyses. For instance, a common task for tools that detect flaky tests [10, 88, 108] (due to test order dependency) is to check that the program state is the same at the beginning of each test.

A common subtask is to find all objects reachable from static variables. Using OGO, we can write a query to get all reachable objects as `query("MATCH (n {[]1}) − [*] → (m) RETURN m", roots)` starting from roots. For another example, OGO could be used to introspect the system, such as finding all objects of a given class that satisfy a desired set of properties. Unlike reflection, which is frequently used to discover relations among objects and meta classes via imperative traversal of object graphs, e.g., serialization code [12, 14, 80, 89, 90], OGO can help find these relations via queries over instances in memory and instances of meta classes (assuming they are available as part of the object graph like in Java).

We discuss these and further benefits in Section 5.

We prototyped OGO for the Java programming language in two ways: (a) by translating an object graph into a Neo4j database on which we run Cypher queries (OGO^{Neo}), and (b) by implementing our own in-memory graph query engine that directly queries the object graph (OGO^{Mem}). The former enables us to harvest the full power of a mature graph database, including the highly optimized query engine and visualization capabilities. However, the translation cost from object graph can be high even with a number of optimizations that we developed, and this approach requires extra memory (disk). Since, at the moment, we do not leverage stored graph databases for any offline analysis, we developed a second prototype that works on the object graph in memory. This approach requires close to zero extra memory, but comes with significant engineering effort.

We evaluate the applicability and robustness of OGO by rewriting 230 assertions from 10 popular open-source projects available on GitHub. Furthermore, we implemented a number of methods from several classes using OGO. We report execution time for both prototypes. Our results highlight substantial performance benefits of in-memory implementation.

The key contributions of this paper include:

- ⋆ **Idea**. OGO introduces a new view of the runtime state of a program and provides a novel way by which such a state can be queried and modified. OGO offers developers a blend of imperative and declarative programming abstractions to manipulate the program state, increasing the expressivity of a programming language which implements OGO's paradigm. Although OGO can be used to replace many statements (even a single field access), it is best suited for tasks that include traversal of objects and metadata, such as introspecting system state, writing assertions and invariants, and implementing linked data structures.
- ⋆ **Formalization**. We formalize OGO by giving a small-step operational semantics to Featherweight Java [49] in terms of property graphs. This formalization captures the core of our translation to Neo4j and can form the foundation for future projects that require reasoning about correctness, such as query optimizations.
- ⋆ **Implementation**. We implement two prototypes of OGO by (a) translating Java's object graph to an off-the-shelf graph database, and (b) by implementing from-scratch-in-memory graph query engine that directly queries the object graph. Although our focus was on features supported by OGO and not on its performance, we describe several optimizations for both translation and in-memory traversal.
- ⋆ **Evaluation**. We evaluated the robustness of our prototypes and compared their performance by rewriting a large number of assertions that are already available in popular open-source projects. Focusing on assertions simplified the selection of target statements for the evaluation and enabled us to scale our experiments. We also implemented a number of methods from several classes in popular open-source projects.

OGO is publicly available at:
https://github.com/EngineeringSoftware/ogo.

2 EXAMPLES

We demonstrate the expressive power of declarative queries for analyzing program state by using two examples, such that each example illustrates a different aspect of the framework: (1) creation of instances, relations between instances and object graph pattern matching; (2) implementing instance methods (`containsKey`) of Java Collections framework class (`java.util.HashMap`).

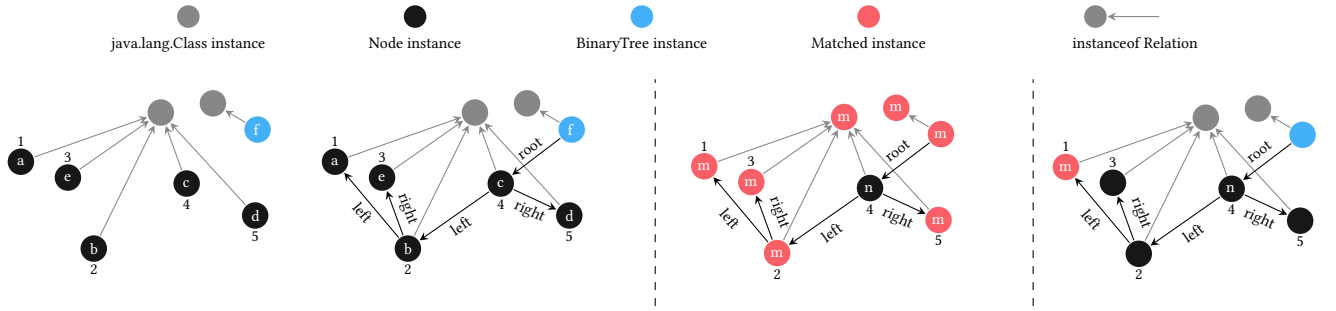
2.1 Creating and Querying Object Graphs

The binary tree is a rooted ordered tree with each of its nodes having at most 2 children. We demonstrate the versatility of OGO by leveraging declarative queries to construct a binary tree and query it for complex patterns. We also use this example to introduce the syntax of the Cypher query language [35].

A Java implementation of the binary tree is given in Figure 5a. An instance of `BinaryTree` contains a reference field `root` of type `Node` (short for `BinaryTree$Node`), the root node of the tree and a primitive `int` field `size` that tracks the total number of nodes in the tree. An instance of `Node` contains references to its left and right child nodes also of type `Node` and stores an integer value in its primitive `int` field `value`.

Constructing a binary tree using OGO is given in Figure 1a. We use the `queryObject` method of OGO to execute the given Cypher query string. Based on the Cypher grammar, the query contains six clauses. The first two are `CREATE` clauses (write to database/object graph), the next three are `MERGE` clauses (write or read from database/object graph) and finally, a `RETURN` clause (defines expressions to be returned).

The first `CREATE` clause (lines 2-3) creates 5 `Node` instances. These instances are assigned variable (which is a term used in the Cypher syntax) names **a-e** for referencing in followup clauses.




```
1 BinaryTree bTree = (BinaryTree) queryObject(
2 "CREATE (a:@1 {value:1}), (b:@1 {value:2}), (c:@1 {value:4}),
3  (d:@1 {value:5}), (e:@1 {value:3})
4 CREATE (f:@2 {size:5})
5 MERGE (a)<-[:left]-(b)<-[:left]-(c)-[:right]->(d) MERGE (b)-[:right]->(e)
6 MERGE (f)-[:root]->(c)
7 RETURN f", Node.class, BinaryTree.class);
```

(a) Creating and connecting `Node` and `BinaryTree` instances.

```
1 ResultSet matchedNodes = query(
2 bTree.root,
3 "MATCH (n {$1})-[*]-(m)
4  RETURN m",bTree.root);
```

(b) Finding all objects reachable from an object.

```
1 ResultSet matchedNodes = query(
2 bTree.root,
3 "MATCH (n {$1})-[:left|right*2]->
4  (m:@1 {value:1})
5 RETURN m", bTree.root);
```

(c) Checking existence of a specific node.

Figure 1: Creating and querying instances and relations between instances using OGO. Graphs above queries visualize the results of those queries. (a) The `CREATE` clause is first used to create instances of `BinaryTree` and `Node` followed by which a `MERGE` clause is used to create relations between the created instances. The reference to the created `BinaryTree` instance (reachable from all instances created by the `CREATE` clause under transitive closure) is returned at the end of the query to prevent the created instances from being garbage collected. (b) The query pattern is undirected and unconstrained in terms of the instance `type` being matched and their distance from n (*root*) and hence all reachable objects are returned. This is particularly useful for identifying object confinement issues. (c) The query pattern is directed and constrained to `Node` instances reachable from n through fields `left` or `right` that are exactly 2 hops away.

The expansion of the positional arguments @1, @2 is described in Table 1 and are replaced with the fully qualified class name (`BinaryTree$Node` and `BinaryTree` respectively) of the arguments following the query string. In Cypher syntax, these are termed `LABELS`. Java non-primitive types are mapped to string `LABELS` of nodes in our graph model. Consequently, the `CREATE` clause creates instances of the specified type. The `LABELS` are followed by node properties ({< `prop_name` >:< `prop_value` >, ...}) which are key-value pairs that map to the represented object's primitive/String fields and their values. In `CREATE`, they assign the fields of the created instance to the specified values. Thus, the `value` field of instances **a-e** is assigned with values 1-5 respectively. The second `CREATE` (line 4) creates an instance of `BinaryTree` (assigned **f**, this can be subsumed into the first `CREATE` but is divided for clarity).

The `MERGE` clauses (line 5) are used to create relationships among the **a-e**. Since the binary tree is a directed graph, we create directed relationships. `LABELS` can also be specified for relationships. In our graph model of the heap, the relationship between a referrer and a referee instance is labelled by the reference field corresponding to the referee instance in the referrer instance's class. We use the reference fields `left` and `right` as labels for the relationships between **a-e**. For e.g., (b) ⟵ [: `left`] – (c) – [: `right`] ⟶ (d) translates to assigning **b** and **d** as the left and right child of **c**. The `RETURN` clause returns a reference to the `BinaryTree` instance **f** to prevent the objects from being garbage collected.

We next describe querying the binary tree by discussing two patterns. The first pattern investigates general reachability of objects from a given object. Such queries are of importance to the problem domains of Aliasing, Confinement and Ownership. For instance, if a node in the binary tree was owned by another instance outside the confinement of the binary tree instance then the aliased node could potentially be mutated, leading to undesirable outcomes.

Figure 1b shows a query which returns all objects reachable from the `root` `Node` instance. The query contains a `MATCH` (line 3) and `RETURN` clauses. The `MATCH` clause matches all paths in the heap's object graph satisfying the given pattern (`n{$1}`) – [∗] – (`m`). The positional argument `$1` expands to a unique identifier belonging to the first argument following the query and is used to uniquely identify an object on the heap. The matched `root` `Node` instance is assigned variable **n**. The pattern neither specifies a label nor a direction for the relationship from **n** and hence all references from and to **n** are considered. Furthermore, the * implies that the referee/referrer instances can be any number of hops away from **n**. Thus, this pattern matches the set of all objects reachable to and from **n** under transitive closure and is assigned **m**. This can be used to identify confinement issues. The pattern given in Figure 1c describes a situation where we are interested in querying the binary tree for existence of a node with a certain value (`value=1`) and a certain distance (2 hops) away from the root node. This is a more constrained query than the former and the pattern (lines 3-4) is more specific. The relationship is now directed (–[] ⟶), labelled and with fixed distance so only instances referenced by **n** and through reference fields `left` or `right` that are 2 ([∗2]) hops away from **n** are considered.

2.2 Implementing Java Library Methods

The Java collections framework provides a rich collection of data structures supported natively by the Java platform. We show how OGO can be used to manipulate these objects by considering the example of implementing methods available in the Java collections' class `java.util.HashMap`. The `HashMap` stores data as key-value pairs where every stored value is mapped to a unique key.

A snippet of the `HashMap` class is given in Figure 2a. The reference field `table` contains all the entries in the map. The method

```
1 public class HashMap<K,V> extends AbstractMap<K,V>
2                           implements Map<K,V>,Cloneable,Serializable {
3     transient Node<K,V>[] table;
4     static class Node<K,V> implements Map.Entry<K,V> {
5         final int hash; final K key; V value; Node<K,V> next;
6     }
7 }
```

(a) Snippet of `java.util.HashMap` class definition.

```
1 public boolean containsKey(Object key) {
2     return getNode(hash(key), key) != null;
3 }
4 final Node<K,V> getNode(int hash, Object key) {
5     Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
6     if ((tab = table) != null && (n = tab.length) > 0 &&
7             (first = tab[(n - 1) & hash]) != null) {
8         if (first.hash == hash && // always check first node
9             ((k = first.key) == key || (key != null && key.equals(k))))
10            return first;
11        if ((e = first.next) != null) {
12            if (first instanceof TreeNode)
13                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
14            do {
15                if (e.hash == hash && ((k = e.key) == key ||
16                    (key != null && key.equals(k)))) return e;
17            } while ((e = e.next) != null);}
18        } return null;}
```

(b) Imperative implementation of `HashMap`'s `containsKey` method.

```
1 public boolean containsKey(Object key) {
2     return queryBool(
3       "MATCH ({$1})-[:table]->()-[*]->()-[:key]->(n) MATCH (m {$2})
4        WHERE n.`equals`(m) = true RETURN COUNT(n) <> 0", this, key);
5 }
```

(c) OGO implementation of `HashMap`'s `containsKey` method.

Figure 2: Methods from Java collections framework using OGO. (a) The nested static class `Node` stores a key-value pair and the reference field `table` stores all the entries in the map. (b) The imperative implementation of `containsKey` uses the `getNode` method. (c) OGO (OGO^{Mem}) can also be used to invoke instance methods as shown in the `WHERE` clause.

`containsKey` of the `HashMap` class checks if a given key is present in the map. A purely imperative implementation of the `containsKey` method is given in Figure 2b. OGO implementation is shown in Figure 2c. The Cypher query used contains 2 `MATCH` clauses (line 3), the first clause matches all instances reachable from `table` that correspond to the reference field `key` (defined in the static nested class `Node` in `HashMap`) and refers to these set of instances as **n**, the second clause matches the instance passed in as an argument to the `containsKey` method. The `WHERE` clause is used to filter the set **n** based on the result of the `equals` method (overriden or inherited from `java.lang.Object`). If the `equals` method evaluates to `true` for at least one instance in **n** and for **m** then the cardinality of set **n** is non-zero after this clause is completed and hence the `RETURN` clause returns `true`.

The examples show a glimpse of the potential of OGO: it provides access to any object in memory, at any point regardless of access specifiers, through a declarative API. Similar to this example, one could envision numerous other potential applications of OGO, a point which we return to in Section 5.

3 FRAMEWORK

We first present OGO's API design (Section 3.1) followed by a high-level overview of the workings of OGO (Section 3.2). Next, we formally describe mapping of the object graph in the JVM heap memory to a property graph, as well as translation to Neo4j (Section 3.3). Lastly, we conclude the section with a description of our implementation and optimizations details (sections 3.4 and 3.5).

```
1 public static ResultSet query(Object root, String fmt, Object... values);
2 public static ResultSet query(String fmt, Object... values);
3 public static Object queryObject(Object root, String fmt, Object... values);
4 public static Object queryObject(String fmt, Object... values);
5 public static boolean queryBool(Object root, String fmt, Object... values);
6 public static boolean queryBool(String fmt, Object... values);
7 public static long queryLong(Object root, String fmt, Object... values);
8 public static long queryLong(String fmt, Object... values);
9 // similar for other primitive types
```

Figure 3: OGO API available via the `OGO` class. Bounded queries (lines 1, 3, 5 and 7) contain an additional root argument that constraints the query execution to a subgraph (limited to only objects reachable from the root argument under transitive closure) of the JVM heap object graph.

3.1 API

We begin by describing queries—their type, arguments, and return values—followed by a discussion of our API design choices.

Queries. We show (the most important parts of) OGO's API in Figure 3. The design choice of keeping the API minimal is intentional, similar to that available for working with relational databases such as `java.sql`. This allows developers acquainted with both Java and Cypher to be able to use OGO with ease.

The highlight of the API are two variadic `query` methods (lines 1 and 2 in Figure 3), which we call *bounded query* and *unbounded query*, respectively. We describe each one in turn.

In case of a *bounded query*, the first argument of the method takes an object (`root`) that constraints query execution only to objects reachable (under transitive closure of reference fields) from this `root`. Having the ability to specify only a subgraph of the entire object heap enables two things: (a) making localized queries, e.g., like we did in Figure 1c, and (b) improve performance of OGO, as we focus on traversal of only a part of the entire object heap. A nice side effect of `root` having an `Object` type is that one can pass a collection that contains multiple roots in a single invocation. For example, if the goal is to find all objects reachable from several static fields, one can add values of all those static fields into a single collection and pass that collection as the first argument to the bounded query.

In case of an *unbounded query*, the query is executed on the entire available object heap. We had an example of such a query in Figure 2c. (In that example an unbounded query was used since the instances being matched may not be related.) Semantics for an unbounded query are relatively less precise than for a bounded query due to the dynamic nature of the JVM, and the user has to accept these unknowns if they use unbounded queries. Specifically, Java is a GC-enabled programming language and OGO *does not* guarantee that objects returned by unbounded queries will not be garbage. In the future, we might offer a more precise semantics for unbounded queries, e.g., ensuring that the result of a query reflects the state as if GC completed its work (this will be trivial to offer during a debugging session for example). Nevertheless, unbounded queries can be valuable in countless examples, e.g., checking if there are instances of classes that are not expected to be instantiated or

Table 1: Positional arguments supported by OGO in the query formatting string.

$	Embed a unique id of an instance
example	`query("MATCH (n {$1}) RETURN n.some_field", node)`
expansion	`query("MATCH (n {__id__:5777203}) RETURN n.some_field")`
@	Embed the fully qualified class name of an instance
example	`query("MATCH (n: @1) RETURN n", node)`
expansion	`query("MATCH (n:`com.package.Node`) RETURN n")`
[]	Embed unique id of instances from an `Iterable` collection and return union of results
example	`query("MATCH (n {[]1}) RETURN n.some_field", nodes)`
expansion	`query("MATCH (n {__id__:8898223}) RETURN n.some_field")`∪ `query("MATCH (n {__id__:5777203}) RETURN n.some_field")`...

checking if at any point all instances of a specific class have a field in a given range of values.

Arguments. Both query methods in our API share the remaining arguments: formatting string (`fmt`) and values. The formatting string in its simplest form is just a Cypher query such as "`MATCH (n : ‘java.util.ArrayList‘) RETURN n`". To enable constraining queries by embedding runtime values, we introduce several kinds of positional arguments; the values are provided from the third argument onward. We show kinds of positional arguments in Table 1. For each kind, we show an example and the expansion once the formatting is complete. We support embedding the unique id of an instance ($), fully qualified class name of an instance (@), or doing a union of query results when we run a query on each element of a (`Iterable`) collection ([]).

Return value. The result of each query is an instance of `ResultSet`. `ResultSet` [85] is available in Java as an interface and a common structure for storing the results of queries; similar structure is used in other programming languages. Via the resulting instance one can access columns (e.g., `getArray(int columnIndex)`), get the current row number (e.g., `getRow()`), etc. Anybody already familiar with working with relational databases from Java would be familiar with the structure.

For convenience, we introduced several query methods that return a specific type (lines 5-8). The only difference is that these methods cast/extract the result as a single value from the `ResultSet`; many assertions or queries that use `COUNT` would end up benefiting from this shorter version. As for the naming, we followed similar convention as the `Unsafe` class [86].

Design decisions. Similar to working with SQL strings and `java.sql`, OGO does not statically type check expressions. Thus, a dynamic `CastClassException` might be raised if a wrong value is passed to one of the query calls. Alternatively, rather than specifying the Cypher query as a string explicitly, the Cypher DSL [67] could be used instead. We leave this integration for future work.

Moreover, our API is not designed to be thread safe. Namely, the developer is responsible to ensure that appropriate locks are held when querying the (sub) object graph. This approach offers more flexibility without being different than implementing any code snippet imperatively.

3.2 Overview of OGO Steps

Figure 4 shows the high level overview of the working of OGO. The figure illustrates steps taken by both of our implemented prototypes (OGO^{Neo} and OGO^{Mem}), and highlights the differences between the two.

OGO flow starts once a query method is invoked, as described in the previous section. In addition to the query, OGO implicitly takes as input the entire state of the program. In the first step, OGO processes the formatting string and builds the actual query to be executed. This step is straightforward and includes simple string manipulations and object id discoveries (if the user has any collection in the formatting string, e.g., [], OGO builds a batch query). In the second step, OGO uses a JVMTI agent (and the root object if given) to identify objects that are in the (sub)graph of interest. Note that a highly optimized system would not traverse the objects before analyzing the given query. Once the second step is done, the execution for OGO^{Neo} and OGO^{Mem} diverge.

OGO^{Mem}, in the third step, builds intermediate representations of the query (AST among others) and executes the query as per the Cypher semantics [34]. In the final step, OGO^{Mem} collects the results into a newly allocated `ResultSet`, which is the final result of the query execution.

Unlike OGO^{Mem}, OGO^{Neo}, in the third step, *translates* (Section 3.3) objects of interest and their relations into a format accepted by Neo4j for batch data loading. In step four, OGO^{Neo} passes the query to a Neo4j database running in a separate JVM and takes the result of the execution, which are node IDs when the result type is non-primitive and primitive values when the result type is primitive. In the final step, it processes these node IDs to build the final result, which are the values known to the JVM. Primitive values remain as they are returned by Neo4j. On the other hand, non-primitive values are mapped to object IDs, which are then used to fetch objects and build the final `ResultSet` returned to the user. OGO^{Neo} and the Neo4j database are implemented as RMI client and server applications respectively. This prevents polluting the JVM heap with irrelevant Neo4j database objects.

3.3 Translation

In this section we describe our translation from the JVM heap to a Neo4j graph database, by formalizing graph databases and presenting an operational semantics for Featherweight Java in terms of this formalization.

Graph databases, formally. A graph database is a *directed multigraph*: a pair $\mathcal{N} \times \mathcal{R}$ of *nodes*, the main entities of the graph, and *relationships*, the edges of the graph denoting directed connections between nodes. A *node* is a pair $\mathcal{L} \times \mathcal{P}$ of a *label*[1], drawn from some abstract domain that serves as the type of the node, and *properties*, a map from string keys to string values. A *relationship* corresponds to the edges of the multigraph: it has a start and end node, a label, and a key-value property map.

The Java heap as a graph database, by example. Following the binary tree example presented in Section 2, consider a simple `BinaryTree` instance in Figure 5b, which defines a `Node1` with a `value` field 4, and a `BinaryTreeb` whose root is a `Node` with 1 as its left subtree, no right subtree and a `value` field of 5. A pictorial representation of the property graph at the marked `POINT` is shown in Figure 5c.

[1] Corresponds to the fully qualified class name of the instance represented by the node.

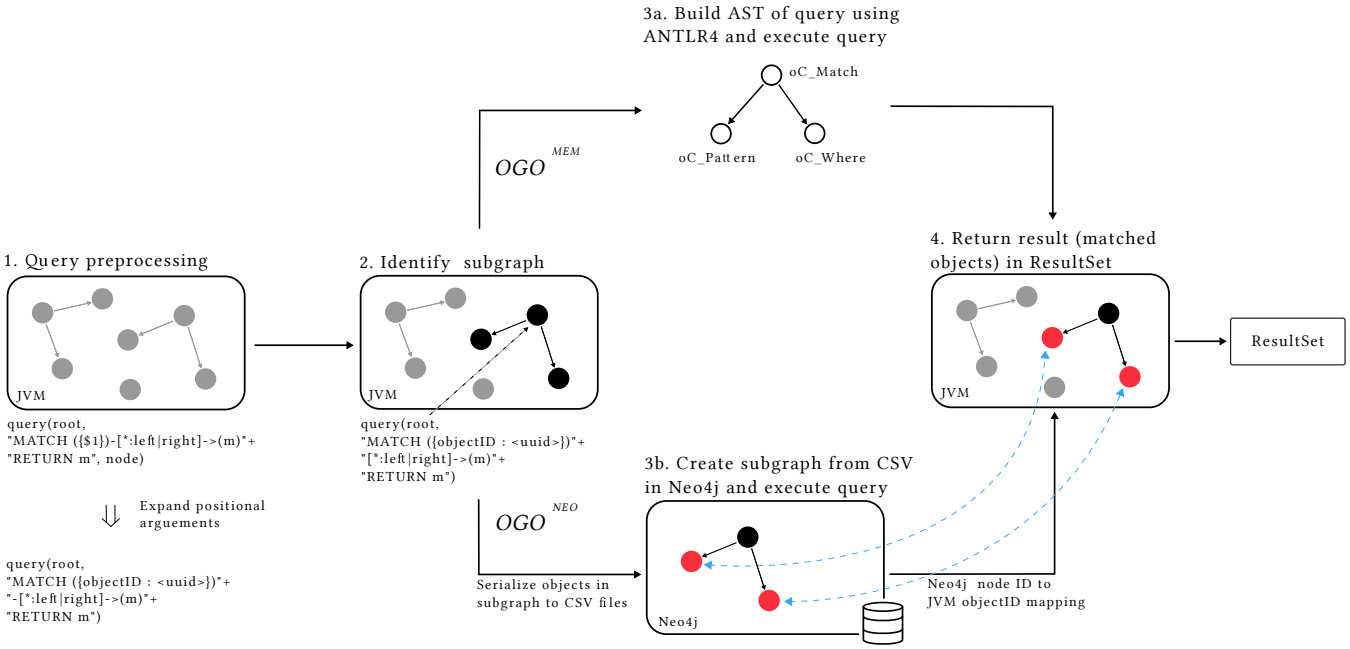


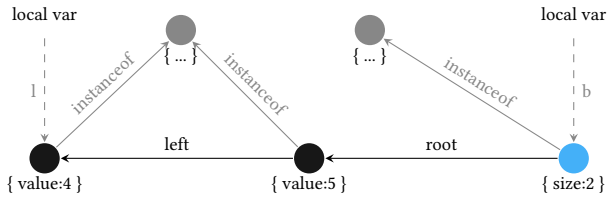
Figure 4: Overview of OGO (and two implementations: OGO^{Neo} and OGO^{Mem}).

```
1 class BinaryTree {
2   private Node root;
3   private int size;
4   static class Node {
5     private Node left;
6     private Node right;
7     int value;
8   }
9 }
```

(a) `BinaryTree` **class definitions.**

```
1 class BinaryTreeTest{
2   void test(){
3     BinaryTree b = new BinaryTree();
4     Node l = new Node(4);
5     b.root = new Node(l, null, 5);
6     (* POINT *)
7     query(...)
8   }
9 }
```

(b) An example test class.



(c) Graph database at ∗ POINT ∗.

Figure 5: An example class, its test class and the corresponding object graph shown as a property graph. (c) instances of `Node`, `BinaryTree` and `java.lang.Class` are shown colored black, blue and grey. The local variable references are shown as dashed edges. Reference fields (`left`,`right`) are mapped as node relations whereas primitive(`value`,`size`) and String fields are mapped as node properties.

- **Object instances.** Every object instance that has been allocated corresponds to a node in the property graph. It's label is the name of the object's class, and it's property set contains the values of any primitive fields or strings. In our example, the node corresponding to `l` has the label `Node` and a singleton property set that maps `value` to 4, while the node corresponding to `b` has the label `BinaryTree` and a singleton property set that maps `size` to value 2.
- **Fields.** Each reference field of an object corresponds to a relationship whose label is the name of the field, its origin is the node of the graph that corresponds to the object it belongs to, its destination is the object corresponds to the field's value, and its property set is empty. The `root` field of `b` is then an edge in the graph going from `b` to the nameless node corresponding to the second allocation, whose `left` field in turn points to `l`.
- **Local variables.** Each local variable we introduce (such as `n` and `l`), gives rise to both a node in the graph whose label is *Local* and whose property set is empty, and to a relationship whose label is the name of the variable from that node to the object corresponding to the local variable's value.
- **Class information.** Each object is also related to a static definition of its class via an `instanceof` relationship. This allows us to capture, for example, static fields belonging to a class as its property set.

$$
\begin{aligned}
&\mathtt{L} ::= \mathtt{class\ C\ extends\ C}\ \{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\\
&\mathtt{K} ::= \mathtt{C}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\mathtt{super}(\overline{\mathtt{x}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{x}};\}\\
&\mathtt{M} ::= \mathtt{C\ m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\mathtt{e}\}\\
&\mathtt{c} ::= \mathtt{x} = \mathtt{new\ C}(\overline{\mathtt{x}}) \mid \mathtt{x.f} = \mathtt{x} \mid \mathtt{x.m}(\overline{\mathtt{x}})\\
&\mathtt{e} ::= \mathtt{c};\ \mathtt{e} \mid \mathtt{return\ x}
\end{aligned}
$$

Figure 6: Syntax of Featherweight Java.

Featherweight Java. Featherweight Java, as introduced by Igarashi et al. [49], constitutes the object-oriented core of Java. Its syntax is shown in Figure 6; it consists of: class declarations such as `class C extends D` $\{\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}$, which introduce a class `C` with `D` as its superclass, $\overline{\mathtt{f}}$ as its fields, `K` as its constructor, and $\overline{\mathtt{M}}$ as its

$$\frac{}{\texttt{return x} \diamond G \longrightarrow^* G} \qquad \frac{\texttt{c} \diamond G \longrightarrow G' \quad \texttt{e} \diamond G' \longrightarrow^* G''}{\texttt{c;e} \diamond G \longrightarrow^* G''}$$

Figure 7: Big-step reduction for Featherweight Java expressions.

$$\frac{(\mathcal{N}, \mathcal{R}) = G \qquad n_{\texttt{x}} = \mathcal{R}(\texttt{x}).3 \qquad n_{\texttt{y}} = \mathcal{R}(\texttt{y}).3 \\ r = (\texttt{f}, n_{\texttt{x}}, n_{\texttt{y}}, \emptyset) \qquad G' = (\mathcal{N}, r \cup \mathcal{R}/\mathcal{R}(\texttt{f}, n_{\texttt{x}}, \cdot))}{\texttt{x.f = y} \diamond G \longrightarrow G'}$$

$$\frac{\mathit{mbody}(\texttt{m}) = \overline{\texttt{y}}.e \qquad \texttt{e}[\overline{\texttt{x}}/\overline{\texttt{y}}, \texttt{x/this}] \diamond G \longrightarrow^* G'}{\texttt{x.m}(\overline{\texttt{x}}) \diamond G \longrightarrow G'}$$

$$\frac{n_{\texttt{x}} = (\mathit{Local}, \emptyset) \qquad n_{\texttt{C}} = (\texttt{C}, \emptyset) \qquad F = \mathit{mkFields}(n_{\texttt{C}}, \texttt{C}(\overline{\texttt{x}}))}{\texttt{x = new C}(\overline{\texttt{x}}) \diamond (\mathcal{N}, \mathcal{R}) \longrightarrow (\mathcal{N} \cup \{n_{\texttt{x}}, n_{\texttt{C}}\}, \mathcal{R} \cup F)}$$

Figure 8: Small-step reduction for Featherweight Java commands.

$$\frac{\mathit{fields}(\texttt{C}) = \overline{\texttt{f}} \qquad \texttt{C}(\overline{\texttt{y}}) = \texttt{this.}\overline{\texttt{f}} = \overline{\texttt{y}}}{\mathit{mkFields}(n_{\texttt{C}}, C(\overline{\texttt{x}})) = \cup\{(f_i, n_{\texttt{C}}, \mathcal{N}(\texttt{x}_i).3), \emptyset\}}$$

$$\frac{\texttt{C extends D} \qquad \mathit{fields}(\texttt{C}) = \overline{\texttt{f}} \\ \texttt{C}(\overline{\texttt{y}}) = \texttt{super}(\overline{\texttt{y}}_1^k)\texttt{; this.}\overline{\texttt{f}} = \overline{\texttt{y}}_k \qquad F = \mathit{mkFields}(n_{\texttt{C}}, D(\overline{\texttt{x}}_1^k))}{\mathit{mkFields}(n_{\texttt{C}}, C(\overline{\texttt{x}})) = F \cup \{(f_i, n_{\texttt{C}}, \mathcal{N}(\texttt{x}_{i+k}).3), \emptyset\}}$$

Figure 9: Calculation of fields relationships.

methods; constructor declarations $\texttt{C}(\overline{\texttt{C}}\ \overline{\texttt{x}})\{\texttt{super}(\overline{\texttt{x}})\texttt{; this.}\overline{\texttt{f}} = \overline{\texttt{x}}\texttt{;}\}$ which initialize the fields of an instance of C; and method declarations $\texttt{C m}(\overline{\texttt{C}}\ \overline{\texttt{x}})\{\texttt{e}\}$ which define a method m with arguments $\overline{\texttt{x}}$ of types $\overline{\texttt{C}}$ whose body e returns a type C. This setup allowed Igarashi et al. to resolve field (*fields*), method type (*mtype*), and method body (*mbody*) lookups from a fixed class table in a straightforward manner, which we will assume in the rest of the presentation.

Unlike Featherweight Java, where the bodies e are a single `return` expression, we consider method bodies in assignment normal form (ANF), as our aim is to formalize the small-step impact of expressions as a graph rather than their big-step reductions and their interactions with (sub)typing. Method bodies are therefore either `return` expressions that return a variable, or sequences of commands c that either allocate a new object, assign to a field, or invoke a method.

What we are ultimately interested in is modelling the shape of the object graph after each command. To that end, we introduce two (mutually recursive) judgements: a small-step judgement that relates a command c and an input graph G to the resulting graph G' and a big-step judgement that relates an entire expression e and an input graph G to the resulting graph G':

$$\texttt{c} \diamond G \longrightarrow G' \qquad \texttt{e} \diamond G \longrightarrow^* G'.$$

The latter is unsurprising (Figure 7), as expressions are either returns (with no effect on the graph) or sequences of commands (which compose the effects of the individual commands). The former is much more intricate, as it actually involves manipulating the object graph via adding nodes, relationships, and labels (Figure 8).

To assign a variable y to the f field of a variable x in a graph $G = (\mathcal{N}, \mathcal{R})$, we first lookup the relationships $\mathcal{R}(y)$ and $\mathcal{R}(x)$, whose labels are y and x respectively, and extract the nodes $n_{\texttt{y}}$ and $n_{\texttt{x}}$ corresponding to the objects these fields currently point to in the object graph (the third component of the relationship tuple). We then create a new relationship r whose label is f, pointing from $n_{\texttt{x}}$ to $n_{\texttt{y}}$ with an empty object map. Finally, we update the object graph to include this new relationship while removing the previous relationship corresponding to the f field of x.

To invoke a method m of a variable x with some arguments $\overline{\texttt{x}}$, we first lookup the body of the method (using *mbody* as defined in Igarashi et al.), which is an expression e parameterized by arguments $\overline{\texttt{y}}$. We then substitute $\overline{\texttt{x}}$ for $\overline{\texttt{y}}$ and x for `this` in e and use the big-step judgment for expressions to construct the resulting object graph G'.

Finally, to create a new object C by invoking its constructor with some arguments $\overline{\texttt{x}}$ and asssigning this new object to a fresh local variable x, we need to extend the graph with two new objects, one corresponding to x (whose label is *Local* and whose property map is empty), and one corresponding to the newly created object (whose label is C and whose property map is empty). Then we need to create a collection of field relationships (Figure 9) to account for the initialization of the new object. We do that via a helper meta-function *mkFields*, identifying two cases:

- If C does not extend another class, then its constructor does not involve a call to `super` and is just a sequence of field initializations $\texttt{this.}\overline{\texttt{f}} = \overline{\texttt{y}}$. In this case, we find the object $n_{\texttt{x}_i}$ corresponding to each argument $\texttt{x}_i$ passed into the constructor (it is the third component of the relationship whose label is $\texttt{x}_i$), and construct a new relationship with an empty property set $(f, n_{\texttt{C}}, \mathcal{N}(\texttt{x}_i).3, \emptyset)$.
- If C extends some other class D, then the first k arguments (denoted as $\overline{\texttt{x}}_1^k$) to the constructor will be used to initialize D, while the rest will be used to initialize C's fields. To construct the full set of new relationships, we recursively call *mkFields* for D with the first k arguments, and then augment the resulting set with the initializations for C's fields, calculated as in the base case.

3.4 Implementation

We now describe the implementation of OGO by considering the execution of the Cypher query in Figure 1c. The execution can be divided into the following 4 steps.

(1) Pre-processing queries. Semantically, the query aims to find patterns that contain instances referring to other instances that are 2 hops away, referenced through fields named `left` or `right` and those which contain a primitive `int` field named `value` with value 1. The first step involves processing the query format string to expand the positional arguments using the expansion described in Table 1. The positional arguments $1 and @1 are expanded into a unique identifier and the fully qualified class name for `bTree.root`, respectively. Any function that maps objects bijectively to a set comprised of either, one of the Java primitive data types or `java.lang.String` elements can be used to generate the unique identifier in the expansion of $. This constraint is based on the design of our property graph model where Java primitive and `String` fields are embedded as node properties. We use the hashcode computed using the

`identityHashCode(Objectobj)` [84] method from the `java.lang.System` package as the unique identifier in our implementation. This function is bijective except for some pathlogical scenarios [83], and can be easily replaced with another function in the future.

(2) Triggering subgraph identification. The next step is the identification of a subgraph of objects on the JVM heap memory, of relevance to the query. Since a significant number of objects in the heap may be irrelevant to the query, it is more efficient to index once and store a subgraph of the complete object graph than to re-traverse it for every additional clause in the query. We store the subgraph in semantically corresponding lightweight C++ classes (for e.g., a Java`class` definition is stored in C++ `ClassInfo` class that stores information about the class name, its field names, their types and modifiers).

We use a native agent developed using the JVMTI [81] framework to identify the subgraph. Native agents can be triggered using JVMTI events based on certain actions from within a Java application. The general approach is to perform an action in the Java domain of OGO after pre-processing the query that triggers an event inside JVMTI. The callback provided by JVMTI to service the event can then be used as the entry point to the native agent. We use the exception event, generated when an exception (any instance of `java.lang.Throwable`) is thrown by a Java method to trigger the native agent. We created a dummy exception class `GraphTriggerException`. This exception is thrown from a dummy method (`setupGraph`) after the query pre-processing step. In the JVMTI callback for handling exception events, we monitor if the signature of the method from which this exception is thrown matches that of `setupGraph`. If a match is found then the native agent proceeds with identification of the subgraph.

(3) Identifying subgraphs. OGO uses JVMTI to identify relations between objects and their field properties (name, value, type, etc.). The native agent is implemented in the C++ programming language. The following are the major steps involved in identifying the subgraph.

Tag zero. JVMTI callbacks use *tags* which are primitive long types to refer to objects on the heap. These tags can be modified inside certain JVMTI callbacks and is the preferred way of identifying and tracking relations across objects. However, these tags can also be modified by JVMTI internal processes. Therefore, as an initialization step, we iterate through all objects on the heap and set their tags to 0. Optionally, our optimization–*Force Garbage Collection*–can be enabled to force a GC event prior to tagging to reduce the total number of objects on the heap consequently, reducing the overhead incurred by all the following JVMTI callbacks.

Loaded classes. The next step involves identifying the types of the objects potentially relevant to the query. This helps limit the number of objects to be considered for inclusion into the subgraph. OGO provides an optimization *whitelist* which can be used to flag certain user specified classes whose instances are guaranteed to be included in the subgraph. In addition, OGO also provides a *blacklist* for specifying classes whose instances are to be definitively excluded from the subgraph. Furthermore, in addition to each object being referred to by tags, JVMTI framework also uses tags to identify classes. By initializing all objects to 0 in the *Tag zero* step, we also initialize the tags of all classes to 0 (instances of `java.lang.Class`). When a class is blacklisted, it remains untagged (tag=0) and hence its instances will not be reported in any of the following JVMTI callbacks since all JVMTI callbacks support filters to filter out untagged objects and classes.

Iterate heap assign unique tag. In this step we assign a unique tag to every instance of every tagged class. We also allocate memory to certain bookeeping C++ classes required for storing information about the subgraph. These unique tags help identify relations between objects in the next step.

Follow references. This JVMTI callback traverses the object graph on the JVM heap. It first reports the referrer and referee instances followed by the primitive fields, String fields and array primitive fields of the referrer instance before doing the like for the referee instance. We once again limit the objects reported by this callback by applying filters ensuring that the reported objects as well as their classes are tagged. By default, if no root object is specified, the traversal is started from a set of system classes, JNI globals and other objects used as roots for garbage collection. The optimization *Fix Root Objects* can be used to start the traversal from the specified object. This reduces the overhead of traversing paths irrelevant to the specified query.

Write graph to CSV. This step applies to OGO^{Neo} exclusively. In this step, we serialize the subgraph into CSV files such that they can be batch imported into a Neo4j database.

(4) Executing the query. For the OGO^{Neo} implementation, the exported CSV files are loaded into a Neo4j database and the given query is executed using Neo4j's Cypher engine. For queries involving the return of a primitive/String fields (stored as properties of Neo4j nodes), the result of Neo4j's Cypher engine is the result of the specified query. For queries that involve the return of an object, we use the semantically equivalent Neo4j node returned by Neo4j's Cypher engine to obtain the unique identifier. Followed by this, we use the JNI framework to retrieve the corresponding instance from the JVM heap.

For the OGO^{Mem} implementation, we used the ANTLR4 [5, 87] parser generator to generate the parser and visitor for the openCypher query language [37]. We use the visitor pattern to deduce the semantics and execute the query. In this implementation, there is no overhead of writing the subgraph to CSV or setting up and creating a Neo4j graph mirroring the state of the JVM's heap. The query is executed directly on the subgraph and like former, JNI is used to report the result back to Java.

3.5 Optimizations

To improve the performance of OGO, we introduced 3 optimizations, Whitelist (WL), Force Garbage Collection (FGC), and Fix Root Objects (FRO).

Whitelist (WL). Limits the size of the subgraph by specifying the type of instances to be definitively included (instances reachable from the specified instance types under transitive closure are also included).

Force garbage collection (FGC). Force a garbage collection event to reduce the number of objects on the JVM's heap before performing the steps to identify the subgraph. This decreases the overhead incurred during the JVMTI callbacks.

Fix root objects (FRO). Limits JVMTI`FollowReferences` callbacks to reporting instances that are transitively reachable from the provided root object. The root object is passed as an argument to the query API call (i.e., bounded query).

4 EVALUATION

We evaluated OGO in two parts: (1) by rewriting existing assertion statements available in tests in open-source projects, and (2) by implementing methods from Java data structure libraries. The first part demonstrates the robustness of our system and ease of its integration with large open-source projects while the second part describes its expressive power over a purely imperative approach. Most of the selected projects are supported by large software organizations, such as Apache or Google. This section describes the chosen subjects and our findings.

The OGO queries were benchmarked on a 64-bit Ubuntu 18.04.1 desktop with a 11th Gen Intel(R) Core(TM) i7-8700 @ 3.20GHz and 64GB RAM. We use Java 11.0.16 and Neo4j 4.4.0 for all experiments.

4.1 Results

This section describes our findings.

Re-written assertions. The selected projects, their lines of code (LOC) and OGO query execution times under different introduced optimizations are given in Table 2. All executions are averaged over 50 runs except for the **Naive** case that is averaged over 3 runs due to long execution time. The reported times are all end-to-end and limited to the scope of OGO API methods (includes the object graph construction time, the query time itself and in case of OGO^{Neo}, serialization and clearing of Neo4j database). We do not consider the total test time, as long running tests would then mask the actual cost of queries. All times are in milliseconds.

Columns 5 through 9 report the query times for OGO^{Neo} prototype whereas those for OGO^{Mem} are given in columns 10 through 12. The final column shows the speedup of OGO^{Mem} over OGO^{Neo}. For OGO^{Neo} and OGO^{Mem} we show time with different optimization levels, which were described in Section 3.5. To compute the speedup, we use OGO^{Neo}:+WL+FRO+FGC and OGO^{Mem}:+WL+FGC. The last two rows in the table show the average (**Avg.**) and the total time ($\sum$) across all the assertions from all projects.

OGO^{Neo}:Naive is not usable since this attempts to include every object on the heap into the graph; there is even a case when the entire run crashed as the VM ran out of memory (Geometry). Each individual optimization provides substantial reduction in the execution time over Naive. +WL+FGC provides the biggest reduction as it dramatically reduces the number of objects to be translated. Finally, combining all the optimizations (OGO^{Neo}:+WL+FRO+FGC) together gives the best performance in most cases. Speedup of OGO^{Neo}:+WL+FRO+FGC over OGO^{Neo}:Naive is up to 94%. Looking at the third column, we find that OGO^{Mem}:Naive is substantially faster than even the most optimal OGO^{Neo}. Furthermore, using +WL+FGC improves the times of OGO^{Mem} by half.

In summary, we find that our optimizations are effective. Furthermore, OGO^{Mem} is, at the moment, better suited for writing program statements due to its low cost. However, we still see OGO^{Neo} being very much usable in a debugging environment, where moderate overhead with a large number of features provided by Neo4j can be effectively used.

Implementing library methods. We re-wrote methods of classes from Guava, the Java Collections Framework (JCF) and the Java Universal Network/Graph Framework (JUNG) using OGO. Table 3 shows the average and total lines of code (LOC) and number of characters (NOC) for implementing the selected methods using OGO and a purely imperative approach (which is already available in those libraries). We see that on average, OGO requires 5 and 4 times as less LOC and NOC than its counterpart.

To confirm the validity of our implementation, we executed in total 611 test methods for Guava, 63 for JUNG and 128 for JCF. We executed all the test methods for Guava and JUNG and only the test methods (present in `java.util` package) for modified classes for JCF. While Guava and JUNG both used JUnit as the test runner, JCF used JTReg. We setup the tests by replacing the imperative implementation of the selected methods with the OGO implementation and then executing the tests.

Furthermore, we benchmarked the implemented methods for our two prototypes with different optimizations for random workloads (number of elements in the data structures were randomly varied between 10-500 for every run) and the results are shown in Table 4. The reported execution times are averaged over 200 runs (We also compared the outcome of OGO implementation with imperative implementation for each run for further checking correctness of OGO). We see that OGO^{Mem} once again outperforms OGO^{Neo}, furthermore, we also observe certain instances where test execution times of OGO^{Neo} is significantly larger. Based on the profiled data, this stems from the unpredictability in execution times of node and relationship creations.

A detailed breakdown of the major steps involved in the OGO^{Neo} flow is given in Figure 10. We see that the optimizations **+WL** and **+WL+FRO** are comparable in performance because, although **+FRO** limits the *Follow references* step to reporting only objects transitively reachable from the passed in root object, the entire object graph is still traversed just not reported in the callback as per JVMTI specifications. Steps such as *Tag zero* and *Assign unique tag* that depend on the total number of objects on the heap are greatly impacted by forcing a garbage collection event through **+FGC**.

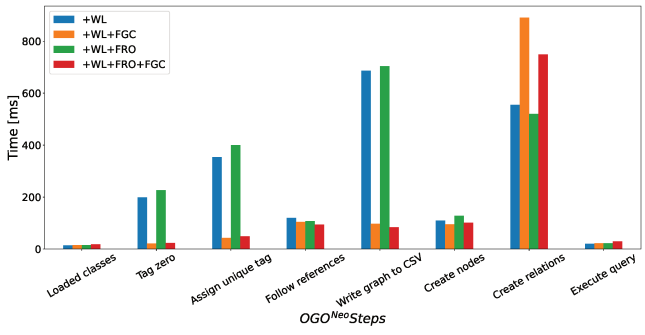


Figure 10: Average time taken by major steps of OGO^{Neo} with the introduced optimizations.

Table 2: End-to-end execution time (ms) of assertions re-written using OGO with different optimizations for external database and in-memory query executions.

Project	SHA	LOC	#Assert	OGO^{Neo}					OGO^{Mem}			Redn.
				Naive	+WL				Naive	+WL		
					-	+FRO	+FGC	+FRO+FGC		-	+FGC	[%]
Assertj-db	8aefa0f	78555	9	832652	7063	7092	4799	4861	851	860	248	95
Cli	0d06c4b	9256	27	99798	1987	1982	1900	1880	224	226	160	91
Csv	1c551d9	9910	12	448403	2867	2713	2291	2171	319	318	178	92
Lang	dded8fd	86960	23	255390	1718	1724	1510	1437	287	286	148	90
Geometry	6c6d046	79996	95	OOM	1407	1397	1321	1267	258	257	145	89
Guava	0ca124d	367861	5	175658	2804	2784	2458	2408	217	218	130	95
Jfreechart	7cdbfbe	134399	10	383694	1811	1824	1625	1548	256	234	134	91
Json-java	31110b5	14829	10	99798	1456	1423	1483	1400	155	155	126	91
Tcases	2fccf74	482581	26	323006	39736	39713	2120	2068	768	796	199	90
Zip4j	fc3a258	15525	18	79658	3528	3548	3322	3157	244	242	174	94
Avg.	-	-	-	269806	6419	6420	2283	2219	358	359	164	-
Σ				2698057	64197	64200	22829	22197	3579	3592	1642	

Table 3: Comparison of LOC(lines of code) and NOC(number of characters) between OGO and imperative implementation of selected instance methods.

	Class	OGO		Imperative	
		LOC	NOC	LOC	NOC
Guava	ArrayTable	9	315	35	1126
	HashBiMap	6	226	26	852
	LinkedListMultiMap	6	202	48	1322
JUNG	SparseGraph	7	303	38	1208
	UndirectedSparseGraph	6	281	36	1066
	DirectedSparseGraph	6	281	36	1078
JCF	ArrayList	6	212	25	426
	HashMap	6	214	37	1073
	LinkedList	6	226	21	374
	ArrayDeque	3	99	15	326
	Vector	6	213	20	413
Avg.	-	6	234	31	842
Σ	-	67	2572	337	9264

5 DISCUSSION AND FUTURE WORK

Although OGO is already production ready, there are endless opportunities for improvements and applications. We document several directions in this section.

Performance. We have focused our work on implementing various features rather than OGO's performance, so far. There is substantial work to be done to get query performance with OGO closer to equivalent imperative implementations. Future work should implement various optimizations that most graph databases already include, such as graph compression techniques [31, 32], indexing [21, 112], memory-efficient custom data structures [43, 105, 106].

Query language. Currently, OGO uses Cypher as the query language. There are several other popular alternatives that can be supported in the future, including Gremlin [95], SPARQL [25], PGQL [104], GSQL [111], and GraphQL [33]. We chose Cypher as it is the most dominant graph query language at this point. Additionally, we were very much familiar with Neo4j.

Debugging. In this paper, we focused the evaluation solely on having OGO being used to program an application. Another direction is to bring OGO to support development tools. In this context, we have preliminarily used OGO to query program state within a debugging session of `jdb` [82]. Our objective was to identify object confinement of the edges and vertices (`Integer`) of a `SparseGraph` (JUNG) instance. We first created a `SparseGraph` instance with one of its vertices being referenced by another object outside the confinement of the `SparseGraph` instance. We next stopped the program execution at a breakpoint using `jdb` and executed an OGO query within the `jdb` session using `jdb`'s `eval` command to check ownership of the vertex. The OGO query was successful in identifying a reference chain to the vertex from the object outside the confinement of the `SparseGraph` instance. Further integration with such debugging environments and IDEs would enable developers to navigate the entire state of a program in an easy way (by writing unbounded queries) and discover interesting values and relations. Finally, having data in a graph database already provides data visualization capabilities with off-the-shelf tools; existing graph visualization libraries are way more advanced than any existing visual debugger [2, 36, 38, 76].

Snapshots. Furthermore, OGO^{Neo} serializes object graphs as a part of its flow and hence, essentially captures a snapshot of the heap. This can be used to analyze differences of the heap. More excitingly, this enables (1) time travel debugging [7, 8, 52], a powerful debugging technique that allows tracking of the sequence of program states leading to the error, and (2) identifying memory leaks [66, 109, 110].

Safety. OGO allows developers to break one of the core software engineering principles: encapsulation. While the power of OGO enables various ways to treat the system, responsible use has a great potential. Furthermore, there are ways in which encapsulation in Java (and other languages) is already being broken (e.g., `Unsafe` [47, 65]) when it comes to designing program analysis tools. Having another, more effective way to implement analyses tools, is a plus.

Languages. OGO idea is applicable beyond Java and integration with other languages, especially those that are dynamically typed, is a planned future work.

6 RELATED WORK

We cover the most closely related work in this section by comparing our work with the following groups: (1) program analysis using query and domain specific languages (DSL), and (2) minimizing impedance mismatch between imperative programming languages and database systems.

Table 4: Comparison of average execution time (ms) for instance methods implemented using OGO for random workloads.

Class	OGO^{Neo} +WL				OGO^{Mem} +WL			
	-	+FRO	+FGC	+FRO+FGC	-	+FRO	+FGC	+FRO+FGC
Vector	1125	794	1254	823	873	316	980	290
UndirectedSparseGraph	7808	2287	24986	8499	1032	186	1045	293
HashBiMap	2470	856	2635	782	519	196	526	179
HashMap	2693	1771	2997	1529	1166	315	1326	289
SparseGraph	13533	2954	16945	2666	1363	188	1444	167
ArrayDeque	1563	1139	1442	1047	1151	320	1104	292
ArrayList	1391	1152	1476	1131	1080	312	1128	292
ArrayTable	5278	1313	5786	1332	624	193	652	178
LinkedList	838	354	3101	827	492	190	638	257
DirectedSparseGraph	3371	3160	10549	3240	1107	188	1215	177
Avg.	4007	1578	7117	2188	941	240	1006	241
Σ	40070	15780	71171	21876	9407	2404	10058	2414

Program analysis. Closely related to OGO are *Fox* [93] and *Datalog* [19]. Fox uses a DSL to analyze object graph in the JVM heap for aliasing, confinement and ownership. Datalog and its applications to (static) program analysis have been explored in numerous studies [16, 48, 55]. Most prominently, the Doop framework [13] and followup work [99] express various forms of pointer analysis in Datalog by exploiting its expressive power. We focus on dynamic program analysis and our key insight that the entire program heap can be seen as a single graph database which can be queried via popular graph query languages. The expressive power of Cypher as a query language enables concise descriptions in applications as shown in Section 2.

Impedance mismatch. *Impedance Mismatch* [62] refers to the friction of interfacing imperative languages with database systems. Efforts to identify, categorize [26, 27, 51] and reduce this mismatch have been achieved through object-oriented databases, object-relational mappers, data access APIs, embedded query languages [26] and language integrated queries.

Call level interface (CLI). are API's such as JDBC [40] and ADO.Net [1, 11, 18, 20] that abstract away the generation of the query language through API methods. However, it is difficult [59] to ensure the efficiency of the generated queries. *JCypher* [97] is an example of a CLI for Cypher.

Embedded query languages. API's such as SQLJ [29, 30] and XJ [41, 42] allow embedding the query language to query external databases. Although OGO shares similarities in that it allows writing queries in Cypher inside Java, however, OGO allows the in-memory object graph to be queried.

Object oriented databases. Object oriented databases (OOD) [6, 17, 56, 63, 101] have been introduced to couple object-oriented programming languages and databases. OOD is more about persistence of objects rather than being able to query relations and object graph available at runtime, which OGO enables.

Relational object mappers. Relational object mappers [15, 23, 60, 79, 103], like Hibernate, enable conversion of data between type systems. They convert objects to (relational) database by automatically grouping properties and enable loading and updating these values. OGO is about querying object graphs not about persistence. The closest connection with object mappers is our translation from an object graph into a graph database.

Language integrated queries. (LINQ) [68], developed by Microsoft, is a technology that adds native data querying capabilities to .NET languages. As a data source, LINQ can use in-memory data, i.e., any collection that implements `IEnumerable` (e.g., `List`, `SortedSet`). Although powerful, LINQ provides no support to query arbitrary objects and their relations. Language integrated queries have seen renewed interest [22, 54, 57, 61, 72, 77, 78, 94, 96, 102] since the release of the LINQ framework. Similar frameworks for Java include SBQL4J [107] and Quaere [75].

7 CONCLUSION

We introduced object graph programming (OGO), a novel paradigm that combines imperative (object-oriented) programming and declarative queries. OGO treats the program state (i.e., object graph) as a graph database that can be queried and modified using graph query language(s); OGO currently uses Cypher as the primary query language. Each object in an object graph is a node, each primitive, String and primitive array field is a property, and each reference field forms a relation between two nodes. OGO is ideal for querying collections (similar to LINQ), introspecting the runtime system state (e.g., finding all instances of a given class or accessing fields via reflection), and writing assertions that have access to the entire program state. We prototyped OGO for Java in two ways: (a) by translating the JVM heap object graph into a Neo4j database on which we run Cypher queries, and (b) by implementing our own in-memory graph query engine that directly queries the object graph. We used OGO to rewrite hundreds of statements in large open-source projects into OGO queries. Our evaluation shows the wide applicability of our approach and good first results with in-memory implementation. OGO enables an entirely different view of objects and data, which will move programming experience to the next level.

ACKNOWLEDGEMENTS

We thank Rifat Seleoglu for his contribution in writing assertions using OGO and Nader Al Alwar, Joseph Kenis, Pengyu Nie, Yu Liu, Zhiqiang Zang, Jiyang Zhang, and the anonymous reviewers for their comments and feedback. This work is partially supported by the US National Science Foundation under Grant CCF-2107206, CCF-2107291, and CCF-2217696.

REFERENCES

[1] Atul Adya, José A. Blakeley, Sergey Melnik, and S. Muralidhar. 2007. Anatomy of the ADO.NET Entity Framework. In *SIGMOD*. 877–888.

[2] Bilal Alsallakh, Peter Bodesinsky, Alexander Gruber, and Silvia Miksch. 2012. Visual Tracing for the Eclipse Java Debugger. In *CSMR*. 545–548.

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *CSUR* (2017).

[4] Renzo Angles and Claudio Gutierrez. 2008. Survey of Graph Database Models. *CSUR* (2008).

[5] ANTLR. 2022. ANTLR Product Website. Accessed November 10, 2022 from https://www.antlr.org.

[6] François Banciihon. 1988. Object-Oriented Database Systems. In *PODS*. 152–162.

[7] Earl T. Barr and Mark Marron. 2014. Tardis: Affordable Time-Travel Debugging in Managed Runtimes. In *OOPSLA*. 67–82.

[8] Earl T. Barr, Mark Marron, Ed Maurer, Dan Moseley, and Gaurav Seth. 2016. Time-Travel Debugging for JavaScript/Node.js. In *FSE*. 1003–1007.

[9] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. 2019. One SQL to Rule Them All - an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. In *SIGMOD*. 1757–1772.

[10] Jonathan Bell and Gail Kaiser. 2014. Unit Test Virtualization with VMVM. In *ICSE*. 550–561.

[11] José A. Blakeley, David Campbell, S. Muralidhar, and Anil Nori. 2006. The ADO.NET Entity Framework: Making the Conceptual Level Real. *SIGMOD Rec.* (2006), 32–39.

[12] Mathias Braux and Jacques Noyé. 1999. Towards Partially Evaluating Reflection in Java. In *PEPM*. 2–11.

[13] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *OOPSLA*. 243–262.

[14] Fabian Breg and Constantine D. Polychronopoulos. 2001. Java Virtual Machine Support for Object Serialization. In *JGI*. 173–180.

[15] Marta Burzańska, Krzysztof Stencel, Patrycja Suchomska, Aneta Szumowska, and Piotr Wiśniewski. 2010. Recursive Queries Using Object Relational Mapping. In *Future Generation Information Technology*. 42–50.

[16] Andrea Calì, Georg Gottlob, Thomas Lukasiewicz, Bruno Marnette, and Andreas Pieris. 2010. Datalog+/-: A Family of Logical Knowledge Representation and Query Languages for New Applications. In *LICS*. 228–242.

[17] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. 1994. A Status Report on the OO7 OODBMS Benchmarking Effort. In *OOPSLA*. 414–426.

[18] Pablo Castro, Sergey Melnik, and Atul Adya. 2007. ADO.NET Entity Framework: Raising the Level of Abstraction in Data Programming. In *SIGMOD*. 1070–1072.

[19] S. Ceri, G. Gottlob, and L. Tanca. 1989. What you always wanted to know about Datalog (and never dared to ask). *Transactions on Knowledge and Data Engineering* (1989), 146–166.

[20] Surajit Chaudhuri, Vivek Narasayya, and Manoj Syamala. 2007. Bridging the Application and DBMS Profiling Divide for Database Application Developers. In *VLDB*. 1252–1262.

[21] Yangjun Chen and Gagandeep Singh. 2021. Graph Indexing for Efficient Evaluation of Label-Constrained Reachability Queries. *TODS* (2021).

[22] James Cheney, Sam Lindley, and Philip Wadler. 2013. A Practical Theory of Language-Integrated Query. In *ICFP*. 403–416.

[23] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-Backed Applications with Query Synthesis. In *PLDI*. 3–14.

[24] Mariano Consens and Alberto Mendelzon. 1993. Hy+: A Hygraph-Based Query and Visualization System. In *SIGMOD*. 511–516.

[25] World Wide Web Consortium. 2013. *SPARQL Protocol and RDF Query Language*. https://www.w3.org/TR/sparql11-query/

[26] William R. Cook and Ali H. Ibrahim. 2005. Integrating programming languages and databases: What is the problem. In *Expert Article*.

[27] George Copeland and David Maier. 1984. Making Smalltalk a Database System. In *SIGMOD*. 316–325.

[28] C. J. Date. 1984. A Critique of the SQL Database Language. *SIGMOD Rec.* (1984), 8–54.

[29] Andrew Eisenberg and Jim Melton. 1998. SQLJ Part 0, Now Known as SQL/OLB (Object-Language Bindings). (1998), 94–100.

[30] Andrew Eisenberg and Jim Melton. 1999. SQLJ Part 1: SQL Routines Using the Java Programming Language. (1999), 58–63.

[31] Wenfei Fan, Jianzhong Li, Xin Wang, and Yinghui Wu. 2012. Query Preserving Graph Compression. In *SIGMOD*. 157–168.

[32] Tomás Feder and Rajeev Motwani. 1991. Clique Partitions, Graph Compression and Speeding-up Algorithms. In *STOC*. 123–133.

[33] The GraphQL Foundation. 2021. *GraphQL Specification*. https://spec.graphql.org/October2021/

[34] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Martin Schuster, Petra Selmer, and Andrés Taylor. 2018. Formal Semantics of the Language Cypher. *arXiv e-prints* (2018), arXiv:1802.09984.

[35] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *SIGMOD*. 1433–1445.

[36] GNU. [n. d.]. DDD - DataDisplay Debugger. Accessed November 10, 2022 from https://www.gnu.org/software/ddd.

[37] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. 2018. openCypher: New Directions in Property Graph Querying. In *EDBT*. 520–523.

[38] Zhongxian Gu, Drew Schleck, Earl T. Barr, and Zhendong Su. 2014. Capturing and Exploiting IDE Interactions. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 83–94.

[39] Marc Gyssens, Jan Paredaens, and Dirk van Gucht. 1990. A Graph-Oriented Object Database Model. In *PODS*. 417–424.

[40] Graham Hamilton, Rick Cattell, and Maydene Fisher. 1997. *Jdbc Database Access with Java: A Tutorial and Annotated Reference* (1st ed.). Addison-Wesley Longman Publishing Co., Inc.

[41] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G. Burke, Rajesh Bordawekar, Igor Pechtchanski, and Vivek Sarkar. 2005. XJ: Facilitating XML Processing in Java. In *WWW*. 278–287.

[42] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G. Burke, Vivek Sarkar, and Rajesh Bordawekar. 2004. XJ: Integration of XML Processing into Java. In *WWW Alt.* 340–341.

[43] Jelle Hellings, George H.L. Fletcher, and Herman Haverkort. 2012. Efficient External-Memory Bisimulation on DAGs. In *SIGMOD*. 553–564.

[44] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. 2021. Knowledge Graphs. *CSUR* (2021).

[45] Florian Holzschuher and René Peinl. 2013. Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4j. In *EDBT*. 195–204.

[46] Florian Holzschuher and René Peinl. 2016. Querying a graph database – language selection and performance considerations. *J. Comput. System Sci.* 82 (2016), 45–68.

[47] Shiyou Huang, Jianmei Guo, Sanhong Li, Xiang Li, Yumin Qi, Kingsum Chow, and Jeff Huang. 2019. SafeCheck: Safety Enhancement of Java Unsafe API. In *ICSE*. 889–899.

[48] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and Emerging Applications: An Interactive Tutorial. In *SIGMOD*. 1213–1216.

[49] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. 1999. Featherweight Java: A Minimal Core Calculus for Java and GJ. In *OOPSLA*. 132–146.

[50] Salim Jouili and Valentin Vansteenberghe. 2013. An Empirical Comparison of Graph Databases. In *SocialCom*. 708–715.

[51] Arthur M. Keller. 1986. Unifying Database and Programming Language Concepts Using the Object Model. In *OODS*. 221–222.

[52] Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2013. Expositor: Scriptable Time-Travel Debugging with First-Class Traces. In *ICSE*. 352–361.

[53] Michael Kifer and Yanhong Annie Liu (Eds.). 2018. *Declarative Logic Programming: Theory, Systems, and Applications*. Vol. 20. Association for Computing Machinery and Morgan & Claypool.

[54] Oleg Kiselyov and Tatsuya Katsushima. 2017. Sound and Efficient Language-Integrated Query. In *Programming Languages and Systems*. 364–383.

[55] Phokion G. Kolaitis and Moshe Y. Vardi. 1990. On the Expressive Power of Datalog: Tools and a Case Study. In *PODS*. 61–71.

[56] Mikael Kopteff. 2008. The Usasge and Performance of Object Databases Compared with ORM Tools in a Java Environment. In *ICOODB*.

[57] Tomasz Marek Kowalski and Radosław Adamus. 2017. Optimisation of language-integrated queries by query unnesting. *Computer Languages, Systems and Structures* (2017), 131–150.

[58] Ravi Kumar, Prabhakar Raghavan, Sridhar Rajagopalan, D. Sivakumar, Andrew Tompkins, and Eli Upfal. 2000. The Web as a Graph. In *PODS*. 1–10.

[59] Ramon Lawrence, Erik Brandsberg, and Roland Lee. 2017. Next Generation JDBC Database Drivers for Performance, Transparent Caching, Load Balancing, and Scale-Out. In *SAC*. 915–918.

[60] Patrick Connor Linskey and Marc Prud'hommeaux. 2007. An In-Depth Look at the Architecture of an Object/Relational Mapper. In *SIGMOD*. 889–894.

[61] J. López-González and Juan M. Serrano. 2020. The optics of language-integrated query. *Science of Computer Programming* (2020).

[62] David Maier. 1990. *Representing Database Programs as Objects*. 377–386.

[63] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. 1986. Development of an Object-Oriented DBMS. (1986), 472–482.

[64] Sofia Maiolo, Lorena Etcheverry, and Adriana Marotta. 2020. Data Profiling in Property Graph Databases. *JDIQ* (2020).

[65] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. 2015. Use at Your Own Risk: The Java Unsafe API in the Wild. In *OOPSLA*. 695–710.

[66] Evan K. Maxwell, Godmar Back, and Naren Ramakrishnan. 2010. Diagnosing Memory Leaks Using Graph Mining on Heap Dumps. In *KDD*. 115–124.

[67] Gerrit Meier and Michael Simons. 2020. The Neo4j Cypher-DSL. https://neo4j-contrib.github.io/cypher-dsl/current/.

[68] Erik Meijer, Brian Beckman, and Gavin Bierman. 2006. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *SIGMOD*. 706.

[69] Jim Melton. 2006. *Database Language SQL*. Springser Berlin Heidelberg, 105–132.

[70] Jan Michels, Keith Hare, Krishna Kulkarni, Calisto Zuzarte, Zhen Hua Liu, Beda Hammerschmidt, and Fred Zemke. 2018. The New and Improved SQL: 2016 Standard. *SIGMOD Rec.* (2018), 51–60.

[71] Aleksandar Milicevic, Derek Rayside, Kuat Yessenov, and Daniel Jackson. 2011. Unifying Execution of Imperative and Declarative Code. In *ICSE*. 511–520.

[72] Fabian Nagel, Gavin Bierman, and Stratis D. Viglas. 2014. Code Generation for Efficient Query Processing in Managed Runtimes. *Proceedings of the VLDB Endowment* (2014), 1095–1106.

[73] Inc. 2022. Neo4j. 2022. Neo4j Product Website. Accessed November 10, 2022 from https://neo4j.com/.

[74] Pengyu Nie, Marinela Parovic, Zhiqiang Zang, Sarfraz Khurshid, Aleksandar Milicevic, and Milos Gligoric. 2020. Unifying Execution of Imperative Generators and Declarative Specifications. (2020).

[75] Anders Norås. 2011. Quaere. Accessed November 10, 2022 from https://github.com/anoras/Quaere.

[76] Rainer Oechsle and Thomas Schmitt. 2002. JAVAVIS: Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI). In *Software Visualization*. Springer Berlin Heidelberg, 176–190.

[77] Atsushi Ohori and Katsuhiro Ueno. 2011. Making Standard ML a Practical Database Programming Language. In *ICFP*. 307–319.

[78] Rui Okura and Yukiyoshi Kameyama. 2020. Reorganizing Queries with Grouping. In *GPCE*. 50–62.

[79] Elizabeth J. O'Neil. 2008. Object/Relational Mapping 2008: Hibernate and the Entity Data Model (Edm). In *SIGMOD*. 1351–1356.

[80] Oracle. 2022. Java Object Serialization Specification. Accessed November 10, 2022 from https://docs.oracle.com/en/java/javase/17/docs/specs/serialization/index.html.

[81] Oracle. 2022. Java Virtual Machine Tool Interface (JVM TI). Accessed November 10, 2022 from https://docs.oracle.com/en/java/javase/11/docs/specs/jvmti.html.

[82] Oracle. 2022. jdb - The Java Debugger. Accessed November 10, 2022 from https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html.

[83] Oracle. 2022. Method Detail: hashCode. Accessed November 10, 2022 from https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#hashCode().

[84] Oracle. 2022. Method Detail: identityHashCode. Accessed November 10, 2022 from https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/System.html#identityHashCode(java.lang.Object).

[85] Oracle. 2022. Package java.sql Description. Accessed November 10, 2022 from https://docs.oracle.com/javase/8/docs/api/java/sql/package-summary.html#package.description.

[86] Oracle. 2022. Unsafe Class. Accessed November 10, 2022 from https://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/sun/misc/Unsafe.java.

[87] T. J. Parr and R. W. Quong. 1995. ANTLR: A predicated-LL(k) parser generator. *Software: Practice and Experience* (1995), 789–810.

[88] Owain Parry, Gregory M. Kapfhammer, Michael Hilton, and Phil McMinn. 2021. A Survey of Flaky Tests. *ACM Trans. Softw. Eng. Methodol.* (2021).

[89] Michael Philippsen and Bernhard Haumacher. 1999. More efficient object serialization. In *Parallel and Distributed Processing*. Springer Berlin Heidelberg, 718–732.

[90] Michael Philippsen, Bernhard Haumacher, and Christian Nester. 2000. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience* (2000), 495–518.

[91] Jaroslav Pokorný. 2015. Graph Databases: Their Power and Limitations. In *Computer Information Systems and Industrial Management*. 58–69.

[92] Jaroslav Pokorný. 2016. Conceptual and Database Modelling of Graph Databases. In *IDEAS*. 370–377.

[93] Alex Potanin. 2002. A Tool for Ownership and Confinement Analysis of the Java Object Graph. In *OOPSLA*. 118–119.

[94] Wilmer Ricciotti and James Cheney. 2021. Query Lifting. In *Programming Languages and Systems*. 579–606.

[95] Marko A. Rodriguez. 2015. The Gremlin Graph Traversal Machine and Language (Invited Talk). In *Symposium on Database Programming Languages*. Association for Computing Machinery, New York, NY, USA, 1–10.

[96] Guido Salvaneschi, Mirko Köhler, Daniel Sokolowski, Philipp Haller, Sebastian Erdweg, and Mira Mezini. 2019. Language-Integrated Privacy-Aware Distributed Queries. *Proc. ACM Program. Lang.* (2019).

[97] Wolfgang Schuetzelhofer. [n. d.]. JCypher. Accessed September 19, 2023 from https://github.com/Wolfgang-Schuetzelhofer/jcypher.

[98] Philipp Seifer, Johannes Härtel, Martin Leinberger, Ralf Lämmel, and Steffen Staab. 2019. Empirical Study on the Usage of Graph Query Languages in Open Source Java Projects. 152–166.

[99] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Datalog Reloaded*. 245–251.

[100] Disha Soni, Thanaa Ghanem, Basma Gomaa, and Jon Schommer. 2019. Leveraging Twitter and Neo4j to Study the Public Use of Opioids in the USA. In *GRADES-NDA*.

[101] David D. Straube and M. Tamer Özsu. 1990. Queries and Query Processing in Object-Oriented Database Systems. *ACM Transactions on Information Systems* (1990), 387–430.

[102] Kenichi Suzuki, Oleg Kiselyov, and Yukiyoshi Kameyama. 2016. Finally, Safely-Extensible and Efficient Language-Integrated Query. In *PEPM*. 37–48.

[103] Alexandre Torres, Renata Galante, Marcelo S. Pimenta, and Alexandre Jonatan B. Martins. 2017. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Information and Software Technology* (2017), 1–18.

[104] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: A Property Graph Query Language. In *International Workshop on Graph Data Management Experiences and Systems (GRADES '16)*. Association for Computing Machinery, New York, NY, USA, 6 pages.

[105] Sebastiaan J. van Schaik and Oege de Moor. 2011. A Memory Efficient Reachability Data Structure through Bit Vector Compression. In *SIGMOD*. 913–924.

[106] Jeffrey Scott Vitter. 2001. External Memory Algorithms and Data Structures: Dealing with Massive Data. *CSUR* (2001), 209–271.

[107] Emil Wcisło, Piotr Habela, and Kazimierz Subieta. 2011. A Java-Integrated Object Oriented Query Language. In *Informatics Engineering and Information Science*. 589–603.

[108] Anjiang Wei, Pu Yi, Tao Xie, Darko Marinov, and Wing Lam. 2021. Probabilistic and Systematic Coverage of Consecutive Test-Method Pairs for Detecting Order-Dependent Flaky Tests. *TACAS* (2021), 270–287.

[109] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *ICPE*. 273–284.

[110] Markus Weninger, Lukas Makor, Elias Gander, and Hanspeter Mössenböck. 2019. AntTracks TrendViz: Configurable Heap Memory Visualization Over Time. In *ICPE*. 29–32.

[111] Mingxi Wu and Alin Deutsch. 2019. *GSQL: An SQL-Inspired Graph Query Language*. https://info.tigergraph.com/gsql

[112] Xifeng Yan, Philip S. Yu, and Jiawei Han. 2005. Graph Indexing Based on Discriminative Frequent Structure Analysis. *TODS* (2005), 960–993.