In-Memory Object Graph Stores

Aditya Thimmaiah
The University of Texas at Austin, TX, USA

Zijian Yi
The University of Texas at Austin, TX, USA

Joseph Kenis
The University of Texas at Austin, TX, USA

Christopher J Rossbach
The University of Texas at Austin, TX, USA

Milos Gligoric
The University of Texas at Austin, TX, USA

Abstract

We present a design and implementation of an in-memory object graph store, dubbed ϵStore. Our key innovation is a storage model – epsilon store – that equates an object on the heap to a node in a graph store. Thus any object on the heap (without changes) can be a part of one, or multiple, graph stores, and vice versa, any node in a graph store can be accessed like any other object on the heap. Specifically, each node in a graph is an object (i.e., instance of a class), and its properties and its edges are the primitive and reference fields declared in its class, respectively. Necessary classes, which are instantiated to represent nodes, are created dynamically by ϵStore. ϵStore uses a subset of the Cypher query language to query the graph store. By design, the result of any query is a table (ResultSet) of references to objects on the heap, which users can manipulate the same way as any other object on the heap in their programs. Moreover, a developer can include (transitively) an arbitrary object to become a part of a graph store. Finally, ϵStore introduces compile-time rewriting of Cypher queries into imperative code to improve the runtime performance. ϵStore can be used for a number of tasks including implementing methods for complex in-memory structures, writing complex assertions, or a stripped down version of a graph database that can conveniently be used during testing. We implement ϵStore in Java and show its application using the aforementioned tasks.

2012 ACM Subject Classification Software and its engineering

Keywords and phrases Object stores, Graph stores, Cypher

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.30

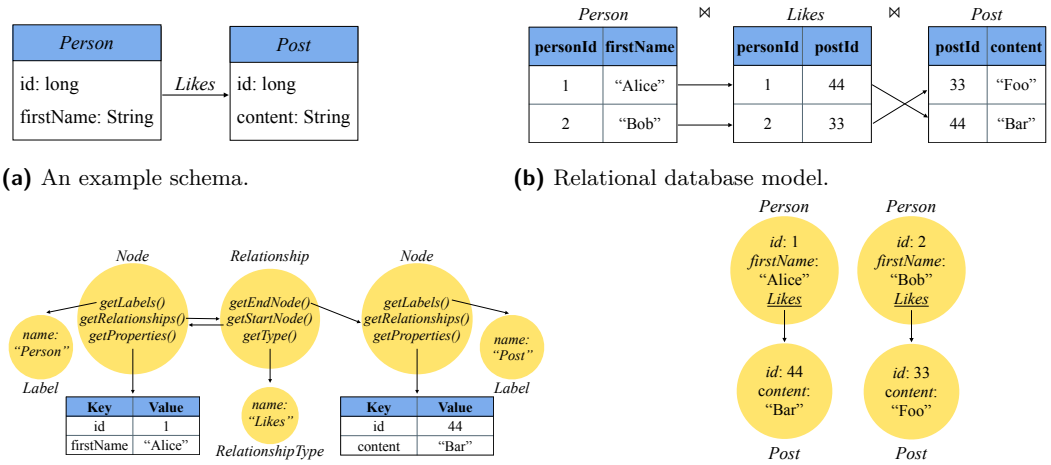
Supplementary Material *Software (Source Code)*: `https://github.com/EngineeringSoftware/eStore`, archived at `swh:1:dir:72a974d02f33141537835dcdb9d1661af263a253`

Funding This work is partially supported by the US National Science Foundation under Grant Nos. CCF-2107291, CCF-2217696, CCF-2313027, and CCF-2403036.

Acknowledgements We thank Michael Y. Levin, Zhiqiang Zang, Yu Liu, Nader Al Awar, Jiyang Zhang, Linghan Zhong, Cheng Ding, Ivan Grigorik, Tong-Nong Lin and the anonymous reviewers for their feedback on this work.

1 Introduction

We present a design and implementation of an *in-memory object graph store*, dubbed ϵStore, which enables easy implementation of methods for complex structures, writing assertions over a set of objects on the heap, and substituting (as a lightweight alternative) a graph database in an application during the testing process.



(a) An example schema.

(b) Relational database model.

(c) Traditional graph database model.

(d) ϵStore model.

Figure 1 Comparison of different storage models for the schema in (a).

ϵStore introduces a novel storage model–epsilon storage–that equates an object on the heap to a node in a graph store. Figure 1 illustrates the differences among well-known storage models and epsilon storage; a schema (or class diagram) that is being instantiated is shown in Figure 1a. In ϵStore, any object on the heap (without changes) can be a part of a graph store (or even multiple stores), and vice versa, any node in a graph store can be accessed like any other object on the heap. Specifically, each node in a graph is an object (i.e., instance of a class). Its properties and its edges are the primitive and reference fields declared in its class, respectively. Necessary classes, which are instantiated to represent nodes, are created dynamically by ϵStore.

ϵStore uses a subset of the Cypher query language [22] to query the graph store; Cypher is a powerful yet concise declarative language popular in the space of graph databases [55]. In our design, the result of any query is a table (ResultSet [51]) of references to objects on the heap, which users can manipulate the same way as any other object on the heap in their programs. Moreover, a developer can include (transitively) an arbitrary object to become a part of a graph store. Processing, e.g., parsing, Cypher queries at runtime can be costly, thus, ϵStore includes support for query rewriting into imperative code at compile-time. Our experiments show query rewrites improving query end-to-end execution times by 5x.

We implemented ϵStore in Java. Our primary focus was to enable novel programming style and propose an efficient storage model. We demonstrate the uniqueness of ϵStore with three use cases. First, we demonstrate the use of ϵStore for concisely expressing complex assertions. Second, we show how ϵStore can be used for implementing various methods. Visualisation of object relations and their fields through graphs can simplify API design. We write methods for a dozen of widely-used data structures from popular libraries (e.g., Guava [21]), as well as methods for H2 [23], an in-memory relational database. We compare ϵStore with OGO [57] in terms of runtime performance. OGO is a framework for Java that allows using Cypher to query the heap. Third, we use ϵStore as a lightweight replacement for graph databases; using an in-memory database (or another form of an object store) is common during testing to save setup cost and runtime cost incurred if a full-blown database is used [6]. At the same time, we note that ϵStore is *not* meant to replace graph databases in production as that is not the primary intent for ϵStore. To estimate benefits of using

ϵStore instead of a graph database, if so desired, we use queries and datasets from the Social Network Benchmark (SNB) [1] of the Linked Data Benchmark Council (LDBC) [56]. LDBC is the most popular benchmark for graph databases. The LDBC SNB was designed to model a snapshot of the activity in a realistic social network. Finally, we evaluate the compile-time code rewriting capability of ϵStore, on the LDBC queries, by comparing its performance with the vanilla version of ϵStore.

Our results show versatility of ϵStore for various tasks and good performance in case a lightweight store is sufficient in the testing process.

The key contributions of this paper include:

- **Idea.** We introduce a novel storage model. Besides being used as a traditional object store (manipulated only via queries), our design enables a unique interoperability between imperative code and objects in a graph store. Results of queries are references to objects in a graph store, thus enabling further imperative processing of the results. Furthermore, any object, which is created by imperative code, can be included into a graph store without any intermediate abstraction and queried for complex relations.
- **Formalization.** We formalize the core of the proposed storage model and the set of API operations supported by ϵStore. We also define a mapping and describe the way instances of any existing class, map to nodes and edges, and can be queried.
- **Implementation.** We implemented ϵStore in Java thus using Java features to dynamically create and load classes that are necessary to represent nodes and their properties. We focus on enabling novel programming models. We also perform compile-time query rewriting into imperative code to reduce the runtime cost. Our implementation is publicly available on GitHub[1].
- **Evaluation.** We performed a three-pronged evaluation. First, we evaluated ϵStore on queries and datasets from the LDBC SNB. Second, we evaluated the power of ϵStore by comparing its execution of library methods of Java data structures, implemented as Cypher queries, with OGO. Finally, we evaluated the compile-time code rewriting capability of ϵStore on the LDBC SNB benchmark queries.

2 Example

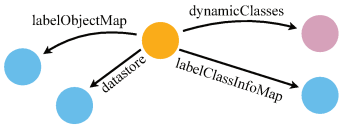
ϵStore is an in-memory graph store. We demonstrate several aspects of ϵStore using an example that showcases the following: (1) creation of a graph store, (2) creation of nodes and edges, (3) querying the graph store, and (4) capturing existing objects into a graph store. We also use this example to provide a brief introduction to the Cypher graph query language [22].

Schema. We use a subset of the LDBC SNB schema shown in Figure 1a to discuss the example. It contains two entities, `Person` and `Post`. The `Person` entity has two properties: `id` of type `long` and `firstName` of type `String`. The `Post` entity has properties: `id` of type `long` and `content` of type `String`. A `Person` can be in a relation (`LIKES`) with a `Post`.

Introduction to Cypher. Cypher is a declarative query language introduced by Neo4j and designed to be expressive when querying graph stores. The labelled property graph (LPG) data model uses nodes and relations to model data. A simple LPG modelling

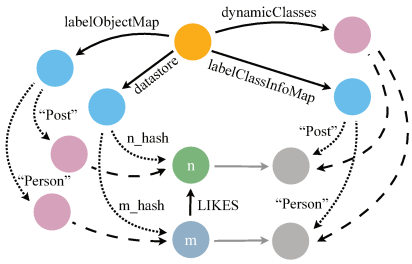
[1] `https://github.com/EngineeringSoftware/eStore`


```
1 /* Instantiate EpsilonStore instance **/
2 EpsilonStore  db = new EpsilonStore ("dbname");
```



(a) Creating an ϵStore instance with name *dbname*.

```
1 /* Add Person and Post nodes to EpsilonStore */
2 db.query(
3  "CREATE (m:`Person` {id: 112, firstName: 'Eve'}"
4  +"-[r:LIKES]->"
5  +"(n:`Post` {id: 481, content: 'About Databases'})"
6 );
```



(b) Insertion of objects without explicit schema (dynamically creating classes `Person` and `Post` at runtime with ASM).

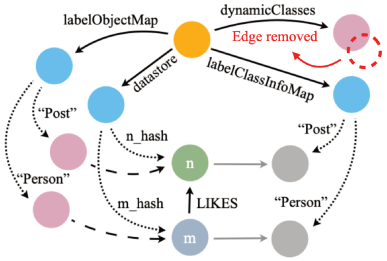
```
1 public class Person {
2     private long id;
3     private String firstName;
4     private Post[] LIKES;
5 }
```

(c) Person.java.

```
1 public class Post {
2     private long id;
3     private String content;
4
5 }
```

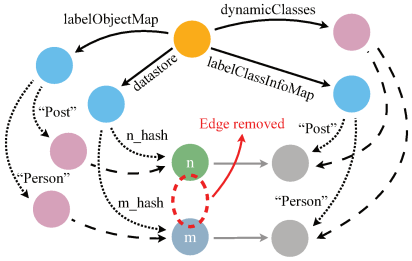
(d) Post.java.

```
1 /* Add Person and Post nodes to EpsilonStore */
2 Post post = new Post(481L, "About Databases");
3 Person person = new Person(112L, "Eve", post);
4 db.captureAll(person);
```



(e) Insertion of objects with pre-defined schema (using existing `Person` and `Post` classes).

```
1 /* Execute Cypher query to remove the LIKES edge */
2 db.query(
3  "MATCH (m:`Person` {id: 112})"
4  +"-[r:LIKES]->"
5  +"(n:`Post` {id: 481}) "
6  +"DELETE r"
7 );
```



(f) Deleting relations using Cypher queries.

Figure 2 An illustrative example to showcase ϵStore for creating a graph store, creating nodes and relations, capturing existing objects, and querying the state of the graph store. The left side of each sub-figure will show the code and the right side will show the corresponding state of the ϵStore instance.

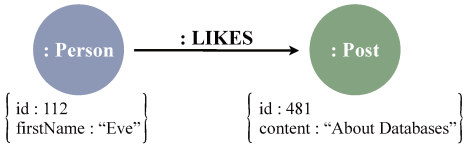


Figure 3 An example labelled property graph.

the schema in Figure 1a is given in Figure 3. In Cypher semantics, the data entities `Person` and `Post` are called *nodes*, and `LIKES` is called a *relation*. Both nodes and relations can have one or more *labels*. Labels function as types, and groups similar nodes and relations. In the example LPG, the `Person` entity has label *Person* and the `Post` entity has label *Post*. The relation between these nodes has the label *LIKES*. Both, nodes and relations, can have properties, which are a set of key-value pairs. The key-value pairs describe the property name and value. The `Person` node has properties *id* with value 112 and *firstName* with value "Eve". A node may have 0 or more relations and can have relations with the same label to different nodes. A relation may be non-directional, uni-directional or bi-directional. We now describe Cypher syntax. A node is generally represented in a Cypher query as `()` and we can optionally specify a variable to reference it in later clauses such as `(n)`. To match and return the node with label `Person`, we would write the Cypher query as `MATCH (n:`Person`{id:112, firstName:"Eve"}) RETURN n`. In a larger context, we can be more specific with the query: `MATCH (n:`Person`{id:112, firstName:"Eve"})-[:LIKES]->(m:`Post`{id:481, content:"About Databases"}) RETURN n`.

Creating the graph store. The ϵStore graph store is created by invoking the constructor of the `EpsilonStore` class with at least one argument to specify the name for the graph store as shown in Figure 2a. The constructor call also instantiates several reference fields of ϵStore. The *datastore* is a map that maps the *ID* of an object present in ϵStore to itself. It can be used to optimize queries through ID based lookups. The *dynamicClasses* is a list that stores the `java.lang.Class` instances created dynamically by ϵStore at runtime. The *labelClassInfoMap* is a map that maps fully qualified class names of classes of objects present in ϵStore to `ClassInfo` instances. The `ClassInfo` class of ϵStore is used to cache primitive and reference field information of a class such as type and name of fields and is used during query execution to access the fields of objects of that class. Finally, the *labelObjectMap* is a map that maps fully qualified names of classes of objects present in ϵStore, to list instances that store objects of the corresponding classes.

Inserting objects without explicit schema. Nodes in ϵStore can be any Java object. Nodes can be created using `CREATE` Cypher queries that instantiate objects of specified classes. If the specified class cannot be found in the classes loaded by the JVM, it is dynamically created by ϵStore. An example of such a query is given in Figure 2b. We refer to such object insertions into ϵStore as insertions *without* explicit schema.

The Cypher query in Figure 2b, as defined by the Cypher grammar, is a *single part query* with an update (`CREATE`) clause. The `CREATE` clause, syntactically, is made up of the token `CREATE` followed by a pattern. The pattern may be a single node pattern (`(n)`) or a multi-node pattern with relations (`(n)-[]->(m)-[]->(p)`) or multiple single node patterns (`(n),(m),(p)`). The pattern in our example is a two node pattern with relations `(:`Person`{...})-[:LIKES]->(:`Post`{...})`. The objective of the `CREATE` clause in ϵStore is to create instances and assign references between them based on the nodes and

relations specified in the pattern. Therefore, the query in Figure 2b creates an instance of type `Person` and an instance of type `Post` with the given properties. It then assigns the reference field `LIKES` of the `Person` instance with the `Post` instance. Assuming that the `Person` and `Post` classes are not present in the classes loaded by the JVM, ϵStore uses the byte code manipulation framework, ASM [7], to first dynamically create these classes.

The *node properties* are mapped to the primitive fields of these classes whereas the relations are mapped to their reference fields. All reference fields, created using class creation, are *Arrays* of type `Object` with the same name as the relation labels. This design decision allows support for one-to-many relations with the same label between nodes.

Inserting objects with explicit schema. ϵStore also supports inserting instances of existing classes using the `captureAll` API method as shown in Figure 2e. `captureAll` captures all references under reflexive transitive closure, and hence, all objects reachable from the captured object are also inserted into the store. The class definitions of `Person` and `Post` are given in Figure 2c and Figure 2d, respectively. The field *dynamicClasses* is empty since the classes of the objects inserted already exist and are not required to be created dynamically.

Data modification through queries. The objects captured into an instance of ϵStore can be modified using Cypher queries. The query (lines 3-6) in Figure 2f deletes the relation between two nodes. It is a single part query with two clauses, a reading (`MATCH`) clause and an update (`DELETE`) clause. The `MATCH` clause identifies a set of objects and their references that matches the specified pattern. The specified pattern is a two node relation pattern specifying the labels, properties and relations for the referrer and referee nodes. Since this pattern exists in our store, the variables `m,r,` and `n`, are mapped to the `Person` instance, the reference field `LIKES` of the `Person` instance, and the `Post` instance respectively. The `DELETE` clause on line 6 deletes all the objects mapped to the variable `r`. Therefore, the field `LIKES` of the referrer node is set to `null` which is shown as the edge being removed in the corresponding graph store state.

3 Graph Store

We first formalize our proposed storage model and the core operations supported by ϵStore (§3.1). We then discuss the mapping between Cypher semantics and Java semantics as implemented by ϵStore (§3.2), followed by a discussion on the API methods provided by ϵStore (§3.3). Finally, we highlight the key implementation details (§3.4).

3.1 Semantics

ϵStore requires no changes to the language syntax, compiler, or execution environment. Thus, we focus on formalizing the core operations of ϵStore such as capturing, deleting, and querying objects. We use big-step operational semantics in our formalization. This section clarifies these operations by providing precise definitions and illustrative examples for each of the rules.

Table 1 shows the key symbols used in our formalization. We define a type (τ) as a (type_name, set_of_fields) pair. Each field is a tuple: (name, type_name), and has a unique name within a type definition. To simplify discussion, we will assume that `int` and `string` are the only primitive types available: (int, $\emptyset$) and (string, $\emptyset$). We define a set type (Set, {}), as an untyped set of values. We also define a metadata type that will be used to describe a type: (Meta, {(name, string)}). We use Γ to denote a set of all available types at runtime.

Table 1 Definitions of key symbols used in our formalization of ϵSTORE operations.

Symbol	Definition
Ξ	The set of all objects available on the heap.
Γ	The set of all available types at runtime.
τ	A type definition ($\tau \in \Gamma$), consisting of a type name and a set of fields.
o	An object on the heap ($o \in \Xi$).
o^τ	A meta-type instance ($o^\tau \in \Xi$).
$\overline{o}$	An instance of ϵSTORE
fields(o)	The set of non-primitive field names belonging to object o.
meta(o)	Retrieves the meta-type instance (o^τ) of the object o.
type(v)	Retrieves the type (primitive) of the primitive value v.
new(τ)	Creates a new object instance of type τ.
ntype(L, f)	Creates a new type named L with a set of fields defined in f which is a tuple of field names and their types ($f = (n_1 : t_1, ..., n_k : t_k)$).
$o[p/v]$	Represents an object o with a set of primitive fields p assigned to a set of values v.
$o.name$	Retrieves the value of the field named $name$ of the object o.
$o.^\star name$	Set of references reachable under transitive closure from the field named $name$ of o via non-primitive fields.
$\Xi(o, f \leftarrow v)$	Assigns the value v to the field f of object o.

We use Ξ to denote all objects available on the heap, i.e., $\Xi = \{o_1, o_2, ..., o_n\}$. Each object ($o$) is an instance of a type (τ) and has a unique identifier. An object has a set of values, each corresponding to one field of the object's type. An access to a field (o.name) returns its current value, and a "star" access to a field ($o.^\star$name) returns a transitive closure, i.e., set of objects reachable via non-primitive fields starting from the given field. We use the following notation to update the field `f` of an object o to value v: $\Xi(o, f \leftarrow v)$.

For each type (τ) in a running program, there is an object (o^τ) on the heap, created by the execution environment, which is an instance of the Meta type (analogous to instance of `java.lang.Class` [47])).

An instance of ϵSTORE is simply an object on the heap ($\overline{o} \in \Xi$). We define the ϵSTORE type as (ϵSTORE, {(store, Set)}). Thus, objects in a graph store are the objects reachable via the store field of an ϵSTORE instance, i.e., $db = \overline{o}.^\star$store, and we have that $db \subset \Xi$. As a result of our design, it is trivial to have any object on the heap inserted into a graph store, to share objects across graph stores, and even to embed one graph store into another.

We define the following helper functions: meta(o) returns the metadata object for the given object; fields(o) returns the set of reference field names for the given object; new(τ) creates a new object (o) of the type τ on the heap and its corresponding set of primitive fields p and their values v is denoted by $o[p/v]$; ntype(L, f) makes a new type (named L) with field names and types defined in the tuple f (tuple of field names mapping to their corresponding types). type(v) returns a name of the primitive type for the given primitive value.

We now formally define the core high-level operations that can be performed on an ϵSTORE instance. In all cases, $o \in \Xi \wedge \overline{o} \in \Xi$ and we use the following configuration:

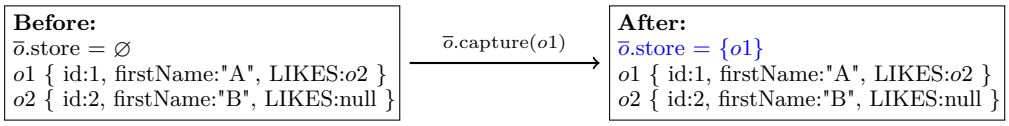
$\langle operation, \Xi, \Gamma \rangle$

For each rule, we first specify its big-step operational semantics, followed by a brief description of the rule, and finally an example showing the state of $\overline{o}$ and other relevant objects, before and after rule application. We assume that all objects used in these examples are instances of type `Person` and that this type exists in Γ unless specified otherwise. For each of the examples, the relevant changes in the after state are highlighted in blue.

▶ **Rule 3.1** (Capture).

$$\frac{\text{store}' = \overline{o}.\text{store} \cup \{o\}}{\langle \overline{o}.capture(o), \Xi, \Gamma\rangle \Downarrow \langle \_, \Xi(\overline{o}, \text{store} \leftarrow \text{store}'), \Gamma\rangle}$$

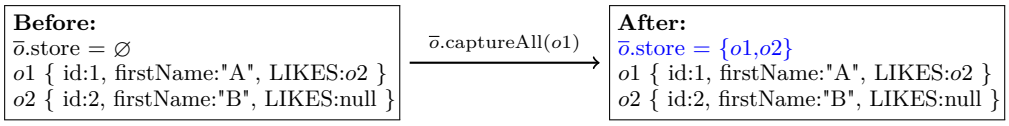
The `capture` rule defines the operation of inserting a single object o (present on the heap Ξ) into an ϵStore instance $\overline{o}$. The union in the premise ensures that if o already exists in $\overline{o}$, then no modifications occur to $\overline{o}$. In the example, we see that the ϵStore instance is empty and there exists an object o on the heap in the before state. After applying the `capture` rule, the ϵStore instance contains the the object $o1$.



▶ **Rule 3.2** (CaptureAll).

$$\frac{C = \overline{o}.\text{store} \cup \{o.^\star f | f \in \text{fields}(o)\} \cup \{o\}}{\langle \overline{o}.captureAll(o), \Xi, \Gamma\rangle \Downarrow \langle \_, \Xi(\overline{o}, \text{store} \leftarrow C), \Gamma\rangle}$$

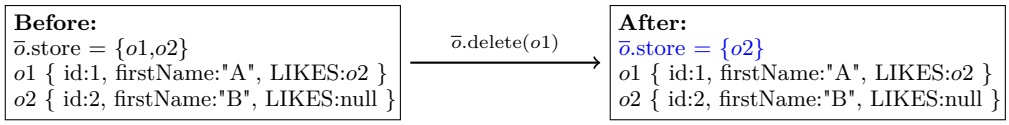
The `captureAll` rule is similar to the `capture` rule but here, the reference fields of the object o being captured are transitively visited and captured into $\overline{o}$ in addition to o. We see that in the premise of the rule, we first collect all the reference fields of o (fields(o)). Next, for every reference field (f), we collect the set of references reachable under transitive closure from o through that field ($o.^\star f$). This set is then unioned with o and the existing store of $\overline{o}$ to get the new store. In the example, we see that the store of $\overline{o}$ is initially empty and there are 2 instances of `Person` ($o1$,$o2$) with one referencing the other. On applying the rule to the referrer instance ($o1$), the store now contains both, the referrer and the referee instances.



▶ **Rule 3.3** (Delete).

$$\frac{\text{store}' = \overline{o}.\text{store} \setminus o}{\langle \overline{o}.delete(o), \Xi, \Gamma\rangle \Downarrow \langle \_, \Xi(\overline{o}, \text{store} \leftarrow \text{store}'), \Gamma\rangle}$$

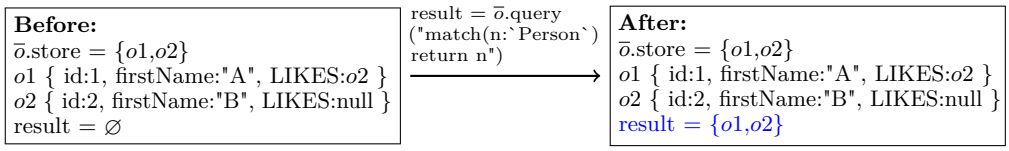
The `delete` rule is used to remove contained objects from ϵStore. In the premise, we remove the object o from the existing store to get the new store (store$'$). This is then used to update the store in the conclusion. If the object does not exist in the store then the store is unmodified after applying the rule. Deleting an object only removes that object from the store while its references that may be part of the store are not removed. In the example, the store initially contains two instances ($o1$,$o2$) with one referencing the other. On applying the rule to the referrer instance, only the referrer instance ($o1$) is removed in the after state of the store.



▶ **Rule 3.4** (Match).

$$\frac{C = \{o \mid o \in \overline{o}.^{\star}\text{store} \wedge \text{meta}(o).\text{name} = L\}}{\langle \_ = \overline{o}.query(\text{“match \{a:}L\text{\} return a”}), \Xi, \Gamma\rangle \Downarrow \langle \_ = C, \Xi, \Gamma\rangle}$$

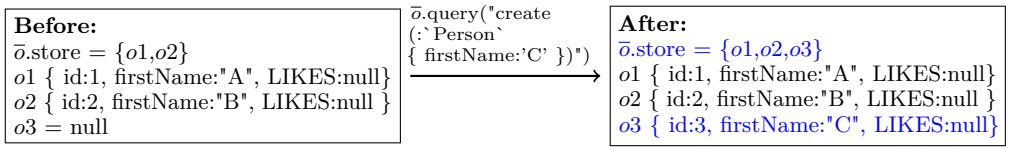
The `match` rule is used to query the store and retrieve stored objects matching one or more specified predicates. Although this rule supports complex predicates, we use a simple predicate to simplify its semantic description. We use the predicate of matching and retrieving all objects in the store whose type matches L. In the premise, we collect the set of all stored objects whose meta-type name matches L. This set is returned as the result of the query in the conclusion. In the example, the store initially contains two instances ($o1$,$o2$) both of type `Person`. The result is initially empty. On applying the rule to retrieve all stored objects of type `Person`, we see that the result now contains $o1$ and $o2$ which are the stored objects of type `Person`.



▶ **Rule 3.5** (Create with pre-defined schema).

$$\frac{L \in \Gamma \qquad o = new(L) \qquad o[p/v]}{\langle \overline{o}.query(\text{“create \{:}L\ \ \{p : v\}\}\text{”}), \Xi, \Gamma\rangle \Downarrow \langle \overline{o}.capture(o), \Xi, \Gamma\rangle}$$

The `create` rule with pre-defined schema is used to create and capture an object of a given existing type into an ϵSTORE instance. This rule uses the `CREATE` clause in the Cypher query. We use a simple example of creating an object of an existing type L with a field p and value v to describe its semantics. In the premise, we first create the object o of type L and update its field named p to value v. In the conclusion, we simply invoke the previously defined `capture` rule on the ϵSTORE instance with o as argument to capture it into the store. In the example below, the store initially contains 2 instances of type `Person` ($o1$, $o2$). After applying the rule, we see that the store now contains 3 instances of type `Person` and $o3$ is now `non-null`. We assign a “name” $o3$ to this created object only for the purpose of showing the before and the after states of applying the rule.

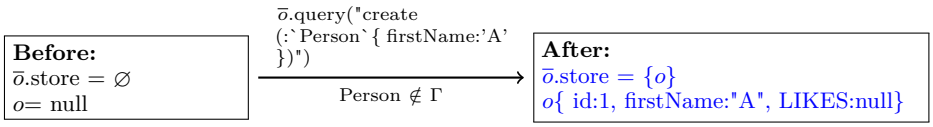


▶ **Rule 3.6** (Create without pre-defined schema).

$$\frac{L \notin \Gamma \qquad \tau = \text{ntype}(L, (\text{“}p\text{”} : type(v))\}) \qquad o = new(\tau) \qquad o[p/v]}{\langle \overline{o}.query(\text{“create \{:}L\ \ \{p : v\}\}\text{”}), \Xi, \Gamma\rangle \Downarrow \langle \overline{o}.capture(o), \Xi, \Gamma \cup \{\tau\}\rangle}$$

The `create` without pre-defined schema is used to create an object of non-existing type and capture it into the ϵSTORE instance. The query used is similar to `create` with pre-defined schema except, now the type L is assumed to not exist in the set of available runtime types

Γ. We use the same example used in create-with-pre-defined-schema rule to describe the semantics. In the premise, we first create a type L with a field named p with type matching that of the primitive value v. Next, we create an instance o of type L and update its field named p with value v. Finally, in the conclusion, we invoke the previously defined `capture` rule on the ϵSTORE instance with o as the argument, to capture it into the store. The newly created type is unioned into the set of available runtime types Γ. This ensures that new objects of this type can now be created by applying the create-with-pre-defined-schema rule instead. In the example below, the store is initially empty. On applying the rule under the assumption that the `Person` type does not yet exist, the store now contains an instance of type `Person` and the object o is `non-null`.



We will now use the operations we formalized to prove 2 properties about ϵSTORE. In the theorems that follow, σ represents the state of the ϵSTORE. $\sigma(o) = \perp$ denotes that the object o is absent from the store. We use the configuration $\langle operation, \sigma, \Gamma\rangle$ in our formal description of the theorems and their proofs.

▶ **Theorem 1** (Idempotency of Insertions). *Repeated insertions of the same object o into an ϵSTORE instance $\overline{o}$ yields an ϵSTORE instance state identical to inserting the object just once. We can formally state the theorem as $\forall\ o$, σ and $\sigma_0(o) \neq \perp$:*

$$\bigwedge_{i=0}^{n} (\langle \overline{o}.capture(o), \sigma_i, \Gamma\rangle \Downarrow \langle \_, \sigma_{i+1}, \Gamma\rangle) \implies \sigma_{n+1} = \sigma_0$$

Equality of the states is defined using the standard definition for set equality i.e., $\sigma_{n+1} = \sigma_0 \iff (\sigma_{n+1} \subseteq \sigma_0 \wedge \sigma_0 \subseteq \sigma_{n+1})$.

Proof. We will use mathematical induction to prove this theorem.

1. **Base Case (i = 0):** Prior to capture, o is present in the store, i.e., $\sigma_0(o) \neq \perp$. When a capture operation is executed, the state transitions as follows:

   $$\langle \overline{o}.\text{capture}(o), \sigma_0, \Gamma\rangle \Downarrow \langle \_, \sigma_1, \Gamma\rangle \qquad \text{where } \sigma_1 = \sigma_0 \cup \{o\} \text{ by rule } \texttt{capture}$$

   σ_1 denotes the updated state. We know that $\sigma_0 \cup \{o\} = \sigma_0$ since $\sigma_0(o) \neq \perp$. Therefore, $\sigma_1 = \sigma_0$ and the theorem holds for the base case.
2. **Inductive Step (i = n):** Consider an $(n-1)^{th}$ transition:

   $$\langle \overline{o}.\text{capture}(o), \sigma_{n-1}, \Gamma\rangle \Downarrow \langle \_, \sigma_n, \Gamma\rangle$$

   Lets assume that the theorem holds for $i = (n-1)$ i.e., $\sigma_n = \sigma_0$. Now for the n^{th} transition, by the semantics defined in rule capture, $\sigma_{n+1} = \sigma_n \cup \{o\}$. However, by our assumption of the theorem holding for $i = (n-1)$, $\sigma_{n+1} = \sigma_0 \cup \{o\}$. By base case, we know that $\sigma_0 \cup \{o\} = \sigma_0$. Therefore, we can claim by mathematical induction that $\sigma_{n+1} = \sigma_0$ as required by the theorem. ◀

▶ **Theorem 2** (Persistence of Inserted Objects). *If an object o is inserted into the store at state σ, then o remains in the store in all subsequent states unless explicitly removed via a deletion operation. We can formally state the theorem as $\forall\ o$, σ and $\sigma_0(o) = \perp$:*

$$(\langle \overline{o}.capture(o), \sigma_0, \Gamma\rangle \Downarrow \langle \_, \sigma_1, \Gamma\rangle) \bigwedge_{i=1}^{n} (\langle S_i, \sigma_i, \Gamma_i\rangle \Downarrow \langle S_i', \sigma_{i+1}, \Gamma_{i+1}\rangle) \implies \sigma_{n+1}(o) \neq \perp$$

where S_i (evaluates to S_i') is any ϵSTORE supported operation excluding the `delete` operation.

Table 2 Mapping of Cypher constructs to Java in ϵStore. "—" indicates unsupported features.

Group	Cypher Construct	Cypher Syntax	Class	
			Dynamically created	Existing
Struct.	Node	`()`	`Object`	`Object`
	Relation	`()-[]-(),()-[]->(),()<-[]-()` `()<-[]->()`	`Object[]` —	`Object`, `Object[]` —
Misc.	Node Label	`(:<label>)`	Fully qualified class name	Fully qualified class name
	Relation Label	`()-[:<label>]-()`	Reference field name	Reference field name
	Node Properties	`({<name>:<value>,...})`	Primitive, String fields	Primitive, String fields
	Relation Properties	`()-[{<name>:<value>,...}]-()`	—	—
Property Values	Decimal Literal	`[Long.MIN_VALUE, Long.MAX_VALUE]`	`long`	`byte`, `short`, `int`, `long`
	Float Literal	`[-Double.MAX_VALUE, Double.MAX_VALUE]`	`double`	`float`, `double`
	Boolean Literal	`TRUE, FALSE`	`boolean`	`boolean`
	String Literal	`"<string>"`	`java.lang.String`	`java.lang.String`

Proof. We will use structural induction on the defined semantic rules to prove this theorem.

1. **Base Case (i = 0):** Prior to capture, o is not present in the store, i.e., $\sigma_0(o) = \bot$. When a capture operation is executed, the state transitions as follows:

$$\langle \overline{o}.\text{capture}(o), \sigma_0, \Gamma \rangle \Downarrow \langle \_, \sigma_1, \Gamma \rangle \qquad \text{where } \sigma_1 = \sigma_0 \cup \{o\} \text{ by rule } \texttt{capture}$$

   σ_1 denotes the updated state. Thus, immediately after execution, o is present in the store or in other words $\sigma_1(o) \neq \bot$. Therefore, the theorem holds for the base case.
2. **Inductive Step (i = n):** Consider an $(n-1)^{th}$ transition:

$$\langle S_{n-1}, \sigma_{n-1}, \Gamma_{n-1} \rangle \Downarrow \langle S'_{n-1}, \sigma_n, \Gamma_n \rangle$$

   Lets assume that the theorem holds for $i = (n-1)$ i.e., $\sigma_n(o) \neq \bot$. Now for the n^{th} transition, the arbitrary statement S_n must be chosen from set of operations including capturing, creating or matching as per the requirements of the theorem. Since these rules do not specify any removal condition and were the same choice of operations available for the $(n-1)^{th}$ transition. We can claim by structural induction on the operational semantics, o persists in all transitions unless explicitly removed through deletion. Therefore we can conclude that $\sigma_{n+1}(o) \neq \bot$ as required by the theorem. ◀

3.2 Mappings

The mapping of semantics between Cypher and Java in ϵStore, is given in Table 2. The first column (Group) shows Cypher features [29] supported by ϵStore. The *Cypher Construct* and the *Cypher Syntax* columns describe the Cypher feature and its corresponding syntax when querying. We show the mapping between Cypher features and Java classes in two cases: (1) classes that are dynamically created by ϵStore (i.e., schema absent), and (2) already existing classes written by developers (i.e., pre-defined schema present). These are given by columns *Dynamically created* and *Existing*, respectively.

A Cypher *node* maps to any object in ϵStore for both dynamic and existing classes. If a *label* is specified for the node, then it maps to objects that are instances of the class with a *fully qualified name* matching the label.

For existing classes, a *relation* with a label maps to objects referred to by other objects with the reference field name matching the label. These referee objects can be elements of an array. For dynamic classes, relations always refer to the elements of an array of objects with the array reference field name matching the label. This allows an object to have relations with the same label to different objects. Relationships also support *directionality* which enforces a referrer-referee relation. ϵStore supports non-directional and uni-directional relations but not bi-directional owing to the limitation of its implementation language (Java).

```
1 public void capture(Object obj) throws EpsilonStoreException
2 public void captureAll(Object obj) throws EpsilonStoreException
3 public ResultSet query(String cQuery) throws EpsilonStoreException
```

Figure 4 ϵStore's API available via the `EpsilonStore` class. The `capture` method in Line 1 is used to capture a single object into ϵStore. The `captureAll` method in Line 2 is used to transitively capture all objects reachable from, and including, the argument `obj` into ϵStore. Line 3 shows the method to query ϵStore with the given Cypher query string `cQuery`.

Node properties map to an object's *primitive* or `java.lang.String` fields in ϵStore. The property values can be a decimal, floating point, string or boolean literal. To handle the range of values supported by Cypher, ϵStore defaults to mapping decimal and float properties to `long` and `double` fields for dynamic classes. Whereas for existing classes, these properties may map to `byte`, `short`, `int`, `long`, `float`, and `double` fields.

Finally, for an instance (o) of a class `C` (`C extends A`), we treat all the fields (those from `C` and `A`) in o the same.

3.3 API

ϵStore's API has three methods: (1) `capture`, (2) `captureAll`, and (3) `query` as shown in Figure 4. It is our intentional design choice to keep the interface simple. Furthermore, the architecture of ϵStore and the expressivity of Cypher allows most operations to be performed through queries.

Inserting data. The `captureAll` method given in Figure 4 line 2 is used to insert data into ϵStore by capturing all objects reachable from the given argument object `obj` under reflexive transitive closure. We use a Breadth First Search (BFS) [10] strategy to traverse the graph of object references reachable from the root object `obj`. `capture` method inserts only the given object into ϵStore.

Semantically, to capture an object in ϵStore translates to storing references to the object in the `EpsilonStore` instance's `labelObjectMap` and `datastore` fields as well as to store its primitive and reference field information (name and type) in `ClassInfo` instance present inside the `labelClassInfoMap` field.

The algorithm for capturing objects into ϵStore is given in Algorithm 1. We use a FIFO queue inside a loop to collect all the objects directly and indirectly reachable from the argument object through its reference fields during each iteration. In each iteration, we pop an object o' from the head of the queue (line 4), compute its hash (ID) (line 8) to check if it already exists in the ϵStore instance (lines 9-11), and add it to `datastore` if absent (line 12). We then get the fully qualified class name of the object's class (line 13), check if its mapped to a list in `labelObjectMap`, and append the object to the list if mapped (line 18). If not mapped, we then insert a new empty list for this key in `labelObjectMap` (line 15) and append the object. In addition, we also create a `ClassInfo` instance for the class of this object to cache its field information and insert it into the `labelClassInfoMap` field (line 16). Following this, we get the reference field objects of o' and insert them into the queue (line 22). The loop terminates when the queue is empty.

Querying. The `query` method is shown in Figure 4 line 3 and is used to query the ϵStore instance. It takes the Cypher query string `cQuery` as an argument and returns a `ResultSet` containing the result of the query: references to objects in the ϵStore instance.

Algorithm 1 Capturing objects.

```
Input: An object o to be inserted into ϵStore
Require: An empty FIFO queue q
 1: procedure captureAll(o)
 2:     Append o to q
 3:     while q is not empty do
 4:         o′ ← Pop head of q
 5:         if o′ is null then
 6:             continue
 7:         end if
 8:         h ← getHashCode(o′)
 9:         if datastore[h] is not null then
10:             continue
11:         end if
12:         datastore[h] ← o′
13:         c ← getClassName(o′)
14:         if labelObjectMap[c] is null then
15:             labelObjectMap[c] ← []
16:             labelClassInfoMap[c] ← new
                ClassInfo(c)
17:         end if
18:         Append o′ to labelObjectMap[c]
19:         cInfo ← labelClassInfoMap[c]
20:         for all r := getRefFields(cInfo) do
21:             o′′ ← getRefObject(cInfo, r, o′)
22:             Append o′′ to q
23:         end for
24:     end while
25: end procedure
```

Algorithm 2 Creating objects.

```
Require: Cypher query string query and list of Java classloaders
         cList
 1: for all x := in Nodes of query do
 2:     l ← Label of x
 3:     pNames ← Property names of x
 4:     pValues ← Property values of x
 5:     pTypes ← inferTypes(pNames, pValues)
 6:     c ← GETORMAKECLASS(l, pNames, pTypes)
 7:     o ← CreateAndSetFields(c, pNames, pValues, pTypes)
 8:     capture(o)
 9: end for

10: procedure getOrMakeClass(l, pNames, pTypes)
11:     if labelClassInfoMap[l] is not null then
12:         return getClass(labelClassInfoMap[l])
13:     else
14:         for all loader := cList do
15:             c ← findClassWithLoader(l, loader)
16:             if c is not null then
17:                 labelClassInfoMap[l] ← new ClassInfo(l)
18:                 return c
19:             end if
20:         end for
21:         c ← ASMCreateClass(l, pNames, pTypes)
22:         labelClassInfoMap[l] ← new ClassInfo(l)
23:         return c
24:     end if
25: end procedure
```

Data can also be inserted into ϵStore through Cypher `CREATE` queries. The algorithm for data insertion through queries is given in Algorithm 2. We describe the case when the Cypher query specifies the creation of a multiple single node pattern (`CREATE (n:label1 {...}), (m:label2), ...`). We start by iterating through all the node definitions in the Cypher query string and collect their labels, property names, property values, and their outgoing edge labels. For each property name and value pair, we infer the property type (line 5) using the mapping given in Table 2. Next, for each node definition, we invoke `GETORMAKECLASS` which either finds a class with a fully qualified name matching that node label or dynamically creates a class with the name matching the label and with its field definitions matching the node's property names and types. The `GETORMAKECLASS` procedure first checks (line 11) the `labelClassInfoMap` field for the class, matching the passed in label argument l and returns it if present (line 12). If absent, we attempt to find the class corresponding to the label by iterating through all the classes loaded into the JVM by all the available *classloaders* [12, 48] and return it if found (line 18). If this also fails, then ϵStore proceeds with dynamic class creation at runtime (line 21). We use the bytecode manipulation and analysis framework ASM [7] to dynamically generate the class. Once the class is found or created, we instantiate it (line 7) and set the fields of the instance to the collected property values. The instance is then captured into ϵStore.

The class instantiation procedure `CreateAndSetFields` checks for consistency of the inferred property types and the field definitions present in the class. An exception is thrown if the checks fail due to a type mismatch.

3.4 Implementation

Field access. Cypher queries may specify patterns in their clauses that require matching a node's properties (`MATCH (n {a:10})`) or its relationships (`MATCH ()-[:label]->()`). Executing these queries requires accessing the fields of objects. To optimize query execution and avoid the overhead of repeatedly retrieving the field name and type for every object,

we cache these field information in ϵStore's `ClassInfo` instances. Since fields are defined in an object's class in Java, it is sufficient to have one `ClassInfo` instance to cache the field information for all objects of that class. These `ClassInfo` instances are stored in ϵStore's `labelClassInfoMap` field. This field is a hashmap that maps a label to its corresponding `ClassInfo` instance. `ClassInfo` is an abstract class. We have two concrete implementations of it based on the approach used to retrieve the field values. We refer to these two implementations of ϵStore as ϵStore^r and ϵStore^u.

- **ϵStore^r** uses Java *reflection* [53] to retrieve the field values. This implementation contains a hashmap mapping field names to their corresponding `java.lang.reflect.Field` [50] instances, obtained using reflection. These `Field` instances are used to retrieve the field values.
- **ϵStore^u** uses Java's *unsafe* [41] API to retrieve the field values. This implementation contains a hashmap mapping field names to field offsets (type `long` values). These field offsets are used to retrieve the field values.

Storage. ϵStore stores inserted objects using their unique ID's and their labels (types).

- **Storing by label**. Every object in Java is an instance of a class which defines its type and the fully qualified name of this class is the *label* of the object. The `labelObjectMap` field is used for storing inserted objects based on their label. This field is a hashmap that maps labels to ordered lists of objects belonging to the corresponding labels. During query execution, if the query string specifies a label for a node to be matched, then since `labelObjectMap` stores objects by their labels, we can use it to efficiently search only a subset of the stored objects.
- **Storing by ID**. The ID is assumed to be unique for every inserted object and is by default computed internally by invoking the `identityHashCode` [46] method provided by `java.lang.System` package on the object. The `datastore` field is used for storing objects based on their ID. This field is a *hashmap* that maps the ID of an inserted object to itself. If the ID of an object being inserted is found to already exist in the `datastore` then the old object reference mapped to that ID in the `datastore` is replaced with the new object reference. During query execution, if the query string specifies a node and its ID then the `datastore` can be used to reduce the search space of stored objects and hence optimize query performance.

Our storage schemes are designed such that the storage structures require minimal update on insertion or removal of objects from ϵStore.

3.5 Code Rewriting

The vanilla version of ϵStore parses input queries and generates a query plan at runtime. We notice the overhead is significant, even multiple times greater than the time to actually execute the query. To reduce overhead, we introduce a code rewriting technique in ϵStore. The basic idea is to parse the query at compile time, then when building the query plan at compile time, we inject the query plan execution code directly into the query call site.

Specifically, we introduce a Java annotation to support this feature. When a method is annotated with this annotation, all the queries (passed as arguments to the `query` API described in Section 3.3) inside the method will be preprocessed by a handler. The handler reuses most part of the ϵStore engine; however, instead of executing the query plan and returning the result, it will aggregate imperative Java code of the query plan needed to be executed for the query and replace the query with corresponding imperative code at the call site. At runtime, only plain imperative Java code is executed. This way, we translate the declarative query into imperative code at compile time, and there is no overhead for parsing and building the query plan at runtime.

```
1 public class LinkedList<E>...{
2     transient Node<E> first; ...
3     private static class Node<E> {
4         ...
5         Node<E> next;
6         Node<E> prev; ...
7     } ...
8 }
```

(a) Snippet of the `LinkedList` class.

```
1 public void testAcyclicity(){
2     List<Long> list = new LinkedList<Long>(); ...
3     EpsilonStore  db = new EpsilonStore ("dbname");
4     db.captureAll(list);
5     assertTrue(db.query(
6       "MATCH (n:`LinkedList$Node`)-[:next*]->(n)"
7      +" RETURN COUNT(n) = 0").getBoolean(0));
8 }
```

(b) Checking *acyclicity* invariant on the `LinkedList` with ϵStore.

Figure 5 An example showing complex assertions with ϵStore.

4 Use Cases

We describe 3 use cases that are made possible as the result of our design. These examples showcase the unique programming style of equating objects (instances of classes) and nodes in ϵStore. Our examples include: (1) writing complex assertions for checking structural invariants, (2) implementing methods, and (3) using ϵStore as a lightweight graph store.

4.1 Runtime invariant checking with complex assertions

Structural invariants can be easily checked by writing complex assertions using ϵStore queries. We demonstrate this by checking the *acyclicity* invariant of a linked list.

A snippet of the `java.util.LinkedList` [13] class definition is given in Figure 5a. It contains an inner-class `Node`, whose instances are the `LinkedList` instance's nodes. The fully qualified class name of this inner-class in Java is `LinkedList$Node`. An instance of `Node` has a field `next` that holds a reference to its successor node in the list. The `next` field of a node can be `null` if it is the last node in the list. It also contains a field `prev` that holds a reference to its predecessor node.

The acyclicity invariant of a `LinkedList` imposes the condition that no node can be reachable from itself by strictly following only its successor or predecessor nodes. In other words, a `LinkedList` must be free of cycles. Figure 5b shows how such an invariant can be checked with ϵStore. The `LinkedList` instance `list` is first captured into an ϵStore instance using its `captureAll` API method (line 4). Next, we assert on the result of the Cypher query (lines 6-7), that checks for acyclicity of the captured `list`. The query contains a `MATCH` clause that matches a pattern in the captured graph of objects, where a `Node` instance contains a path to itself through 1 or more `next` edges. The `RETURN` clause returns `true` if such a pattern does not exist.

In this manner, an otherwise complex assertion can be concisely expressed using Cypher queries with ϵStore.

4.2 Implementing methods with Cypher queries

H2 is an open-source, lightweight relational database implemented in Java. H2 supports embedded in-memory mode, where it runs within the same JVM as the application. Thus, all the objects related to an H2 instance are on the heap. As a result, we can insert an instance of H2 into an instance of ϵStore. We can then query anything related to the H2 instance or data within that instance. Here, we show a way to query the metadata of an H2 instance.

Figure 6a shows how to get the schemas in an H2 database using the API provided by JDBC [52, 25]. Figure 6d shows the actual implementation of the `getSchemas` API by H2. It imperatively setups `result`, iterates over the schemas and insert them into the `result`.

```
Connection conn = DriverManager.getConnection(
    "jdbc:h2:mem:h2TestDb",
    "sa",
    ""
);
DatabaseMetaData meta = conn.getMetaData();
ResultSet schemas = meta.getSchemas();
```

(a) Getting *schemas* using JDBC API.

```
EpsilonStore  db = new EpsilonStore ("name");
Class h2db =
    Class.forName("org.h2.engine.Engine");
db.captureAll(h2db);
/* schemas names are the keys of
 * a ConcurrentHashMap
 */
ResultSet schemas = db.query(
  "MATCH (db: `org.h2.engine.Database`)"
  +"-[:schemas]->()-[:table]->()-[:key]->(k)"
  +"RETURN k");
```

(b) Getting *schemas* using ϵSTORE query.

```
ResultSet users = db.query(
   "MATCH (db: `org.h2.engine.Database`)"
   +"-[:usersAndRoles]->()-[:table]->()"
   +"-[:key]->(k) "
   +"RETURN k");
```

(c) Getting *users* using ϵSTORE query.

```
public ResultInterface getSchemas() {
  return getSchemas(null, null);
}
public ResultInterface getSchemas(String catalog,
  String schemaPattern) {
  checkClosed();
  SimpleResult result = new SimpleResult();
  result.addColumn(
      "TABLE_SCHEM", TypeInfo.TYPE_VARCHAR);
  result.addColumn(
      "TABLE_CATALOG", TypeInfo.TYPE_VARCHAR);
  if (!checkCatalogName(catalog)) {return result;}
  CompareLike schemaLike = getLike(schemaPattern);
  Collection<Schema> allSchemas =
      session.getDatabase().getAllSchemas();
  Value cValue =
      getString(session.getDatabase().getShortName());
  if (schemaLike == null) {
    for (Schema s : allSchemas)
      result.addRow(getString(s.getName()), cValue);
  } else {
    for (Schema s : allSchemas)
      if (schemaLike.test(s.getName()))
        result.addRow(getString(s.getName()),
        cValue);
  }
  // we ignore sorting for a fair comparison
  // result.sortRows(
  // new SortOrder(session, new int[] { 0 }));
  return result;
}
```

(d) H2 implementation for `getSchemas` JDBC API.

Figure 6 Querying metadata of H2. (a) Querying schemas using JDBC `getSchemas` API, (b) Querying schemas of a captured H2 instance with ϵSTORE, (c) Querying users of a captured H2 instance with ϵSTORE, and (d) H2's implementation of `getSchemas`.

Figure 6b shows how to get the same result by inserting the embedded H2 database into ϵSTORE and querying its metadata using Cypher. The idea here is to show how ϵSTORE can be easily used to implement some API methods in a concise and readable way, allowing developers to quickly experiment with new ideas and move fast. As another example, figure 6c shows how we can query all the users in an H2 database while JDBC only provide an API for getting current username.

4.3 Lightweight in-memory Graph Store

The ability to insert, delete, update and query objects in ϵSTORE and the support for the Cypher query language makes ϵSTORE a good candidate for testing when a graph database is needed. We demonstrate in section 5.4 that ϵSTORE can be used as a light-weight alternative in-place of graph databases by evaluating it on the LDBC SNB benchmark.

5 Evaluation

We evaluated ϵSTORE in three ways. First, we benchmarked ϵSTORE on the LDBC SNB [1] benchmark using Neo4j graph database as reference. Second, we re-implemented a number of imperative library methods of data structures using Cypher in ϵSTORE and compared its performance with OGO. Finally, we compared the query execution times of ϵSTORE with and without code rewriting, on the LDBC SNB benchmark. We answer the following questions:

RQ1: How does ϵSTORE perform as a lightweight graph store?
RQ2: How does ϵSTORE, when used for implementing methods, compare with OGO?
RQ3: How does ϵSTORE's code rewriting improve its performance?

We describe environment setup (§5.1), existing systems we use as baselines (§5.2), and the benchmarks (§5.3). Finally, we answer the research questions (§5.4-§5.6).

5.1 Experiment Setup

We built a Docker image for each system used in the evaluation (e.g., OGO) to ensure ease of repeatability of our evaluation experiments. All experiments are run inside Docker containers and averaged over 5 runs. We modified each system used in the evaluation to collect the same profile data. We use a single machine to run the experiments; the machine has an x86_64 11th Gen Intel(R) Core(TM) i7-11700K @ 3.60GHz server with 64GB of RAM and running a 64-bit Ubuntu 20.04.1 operating system. We use Java 17 throughout our experiments.

5.2 Existing Systems

We briefly describe the existing systems that we used in our evaluations.

Neo4j. Neo4j [28] is a graph database and arguably, the most popular one in the industry at the moment. It uses the LPG data model. Neo4j has two modes.

- Server Mode (Neo4j^s): In this mode, Neo4j operates as a database server and runs in a JVM separate from the test JVM which contains the benchmarking queries. We use the Neo4j Java driver [35] version 4.3.3 in the test JVM to send the benchmarking query strings to the Neo4j server. The driver implements the Bolt [30] protocol (similar to JDBC) to communicate with the server. We build Neo4j from source inside docker and load it with the LDBC SNB benchmark dataset. The loading of the datasets (CSVs describing nodes and relations) is done using Neo4j's batch import tool `neo4j-admin` [34].
- Impermanent Mode (Neo4j^i): In this mode, all data inserted into the database are stored in-memory and is non-persistent, and the database runs inside the JVM running the LDBC benchmark queries. This mode is only available in internal test-suites of Neo4j. The Docker image used for evaluation is the same as that built for the server mode. We first create an impermanent database by instantiating `GraphDatabaseService` [31] through dependency injection with `ImpermanentDbmsExtension` [32]. We then insert the LDBC SNB benchmark datasets into the database by using `CREATE` Cypher queries to create the corresponding nodes and relations through database transactions.

We include both, Neo4j^s and Neo4j^i as a point of reference in our evaluation of ϵStore on the LDBC SNB benchmark. We use Neo4j version 5.13.0 and default configuration for all modes of Neo4j in our experiments.

OGO. OGO [57], similar to LINQ [42], combines imperative and declarative (via Cypher) styles of programming. Namely, OGO sees the entire JVM heap (i.e., object graph) as a single graph and enables developers to query the heap (or a subset of it) using queries. We compare it with ϵStore for writing methods using queries on several data structures by replacing existing imperative implementations.

Table 3 shows some major differences between the existing systems we used in our evaluation and ϵStore. We categorize the feature differences into *Programmability* and *Database* features. Programmability features broadly include capabilities such as schema creation, querying runtime program state, manipulating objects on the heap and quickly implementing methods of library classes. These are (partially) supported by OGO and ϵStore. However, most graph databases lack all or most of these features. The database features include some

Table 3 Differences between traditional graph databases, OGO and ϵStore.

	Feature	GDBs	OGO	ϵStore
Programmability	Schema creation	✓	✗	✓
	Query program state	✗	✓	✓
	Heap manipulation	✗	✓	✓
	Method implementation	✗	✓	✓
	Code Rewriting	✗	✗	✓
Database features	Views [43]	✓	✗	✓
	Multi-tenancy [8]	✓	✗	✓
	In-memory	✓	✓	✓
	ACID [24]	✓	✗	✗

features found in traditional databases such as support for multiple views, multi-tenancy, in-memory or non-persistent storage, and ACID compliancy. Most graph databases support all or most of these features, while ϵStore focuses on support for multiple views, in-memory and multi-tenancy features. This table serves to highlight the differences in the design of traditional graph databases and ϵStore and thus their area of applicability.

5.3 Benchmarks

This section provides a brief description of the benchmarks used in our evaluation.

ϵStore as a graph store. To evaluate ϵStore as a graph store and answer RQ1, we use the SNB benchmark from LDBC [56]. LDBC provides both, various sized datasets for its benchmarks and the Cypher queries. The size of a dataset is measured using scale factor which is its uncompressed disk space (e.g., an uncompressed dataset that requires 10GB of disk space would have a scale factor of 10). The LDBC SNB was designed with the aim to model a snapshot of the activity in a realistic social network during a period of time. Table 4 shows the nodes and relationships that appear in the LDBC SNB benchmark and their variation with scale factors used in our evaluation namely, 0.1, 0.3, 1, 3 and 10. Higher scale factors can be supported since ϵStore is only limited by the memory available to the JVM which can be increased with the JVM option `-Xmx`. We observe that the frequency of some nodes (`Comment`) and relationships (`Person Likes Comment`) scale by order of magnitude for an order of magnitude increase in scale factor whereas that of others such as `Tagclass` and `Tagclass IsSubclassOf Tagclass` do not change with scale factor. Query execution time is affected by these different occurrence frequencies depending on the node and relationship labels appearing in it. We use all the queries from the LDBC SNB benchmark that are currently supported by ϵStore. Many queries use Cypher language features which are not implemented in ϵStore yet (§6). Table 5 gives a brief description of the used queries. The name of the query as it appears in the LDBC SNB benchmark documentation is shown in column 1. The queries all contain either a read (`MATCH`) clause or a read and an update (`CREATE`) clause. These read and update clauses contain either single node or two-node patterns. The pattern contained in the queries is given by columns 3, 4 and 5. Finally, a brief description of the queries is given in column 6. Generally, ignoring indexing schemes, we should expect the query execution time to scale with the number of operations performed and the frequency of occurrence of labels in its patterns. For example, Q^G_{SNB} contains the most occurring label (`Comment`) in the benchmark in its patterns and 2 clauses, and would be expected to take more time to execute than Q^D_{SNB} that involves less frequently occurring labels and just 1 clause.

Table 4 Node and relationship statistics for LDBC SNB benchmark across evaluated scale factors.

Type	Name	0.1	0.3	1	3	10
Node	Comment	151043	523222	2052169	6413095	21865475
	Forum	13750	31097	90492	221792	595453
	Person	1528	3514	9892	24328	65645
	Post	135701	324825	1003605	2597141	7435696
	Organisation	7955	7955	7955	7955	7955
	Place	1460	1460	1460	1460	1460
	Tag	16080	16080	16080	16080	16080
	Tagclass	71	71	71	71	71
Relationship	Comment **HasCreator** Person	151043	523222	2052169	6413095	21865475
	Comment **HasTag** Tag	191303	680738	2698393	8426418	28740194
	Comment **IsLocatedIn** Place	151043	523222	2052169	6413095	21865475
	Comment **ReplyOf** Comment	76787	265931	1040749	3251228	11089373
	Comment **ReplyOf** Post	74256	257291	1011420	3161867	10776102
	Forum **ContainerOf** Post	135701	324825	1003605	2597141	7435696
	Forum **HasMember** Person	123268	404952	1611869	4982242	17168614
	Forum **HasModerator** Person	13750	31097	90492	221792	595453
	Forum **HasTag** Tag	47697	108649	309766	767382	2065319
	Person **HasInterest** Tag	35475	81066	229166	569918	1535511
	Person **IsLocatedIn** Place	1528	3514	9892	24328	65645
	Person **Knows** Person	14073	44760	180623	565247	1938516
	Person **Likes** Comment	62225	291590	1438418	5281725	19949360
	Person **Likes** Post	47215	177064	751677	2498139	8839875
	Person **StudyAt** Organisation	1209	2792	7949	19497	52632
	Person **WorkAt** Organisation	3313	7697	21654	53023	143553
	Post **HasCreator** Person	135701	324825	1003605	2597141	7435696
	Post **HasTag** Tag	51118	179499	713258	2229757	7599701
	Post **IsLocatedIn** Place	135701	324825	1003605	2597141	7435696
	Organisation **IsLocatedIn** Place	7955	7955	7955	7955	7955
	Place **IsPartOf** Place	1454	1454	1454	1454	1454
	Tagclass **IsSubclassOf** Tagclass	70	70	70	70	70
	Tag **HasType** Tagclass	16080	16080	16080	16080	16080

ϵStore as a heap manipulation engine. To evaluate ϵStore as an engine to modify objects on the heap and answer RQ2, we use data structures from three sources: Java Collections Framework (JCF) [49], Google Guava [21], and the Eclipse Collections [17] projects. We rewrote on average 2 library methods from each of these data structures to use Cypher queries (rather than the imperative implementation). Simply, for ϵStore, we insert the data structure into an instance of ϵStore and run a query that implements the same functionality as exiting imperative code.

5.4 ϵStore as a Lightweight Graph Store (RQ1)

We use query execution time and memory consumption during benchmarking on LDBC SNB benchmark to motivate ϵStore as a lightweight graph store.

Query execution time. In our early experiments, we noticed substantial variations in query execution times across several runs, which we attribute to the JVM environment. To stabilize the time, we perform the following steps. We first load the dataset for a given scale factor into the systems used in our evaluation. Next, we execute all the chosen LDBC queries for that benchmark in a randomized order. We call this as the 1^{st} set. Following this, we once again execute the same set of queries in another randomized order. We call this as the 2^{nd} set. The sets are executed back-to-back in the same JVM process. The profile data for an evaluation run is collected for every query execution in both the sets. However, when

Table 5 Description of the LDBC SNB benchmark queries used in our evaluation.

Query	Abbrv.	Start Node	Relation	End Node	Description
interactive-delete-query2	Q^A_{SNB}	Person	Likes	Post	Matches a pattern with a *Person* node pointing to a *Post* node through a relation *Likes*, deletes the relation and returns the count of *Likes* relation between the two nodes.
interactive-delete-query3	Q^B_{SNB}	Person	Likes	Comment	Matches a pattern with a *Person* node pointing to a *Comment* node through a relation *Likes*, deletes the relation and returns the count of *Likes* relation between the two nodes.
interactive-delete-query5	Q^C_{SNB}	Forum	HasMember	Person	Matches a pattern with a *Forum* node pointing to a *Person* node through a relation *HasMember*, deletes the relation and returns the count of *HasMember* relation between the two nodes.
interactive-short-query1	Q^D_{SNB}	Person	IsLocatedIn	Place	Matches a pattern with a *Person* node pointing to a *Post* node through a relation *IsLocatedIn* and returns properties of the two nodes.
interactive-short-query5	Q^E_{SNB}	Comment	HasCreator	Person	Matches a pattern with a *Comment* node pointing to a *Person* node through a relation *HasCreator* and returns properties of the *Person* node.
interactive-update-query2	Q^F_{SNB}	Person	Likes	Post	Matches a *Person* and *Post*, creates a relation *Likes* from *Person* to *Post* node and returns the count of *Likes* relation between the two nodes.
interactive-update-query3	Q^G_{SNB}	Person	Likes	Comment	Matches a *Person* and *Comment*, creates a relation *Likes* from *Person* to *Comment* node and returns the count of *Likes* relation between the two nodes.
interactive-update-query5	Q^H_{SNB}	Forum	HasMember	Person	Matches a *Forum* and *Person*, creates a relation *HasMember* from *Forum* to *Person* node and returns the count of *HasMember* relation between the two nodes.
interactive-update-query8	Q^I_{SNB}	Person	Knows	Person	Matches a *Person* and *Person*, creates a relation *Knows* from *Person* to *Person* node and returns the count of *Knows* relation between the two nodes.

reporting the profile data for a query for an evaluation run, we use the data from the 2^{nd} set and discard the 1^{st} set. Figure 7 shows a boxplot of the query execution time for the queries in the 1^{st} and 2^{nd} sets for ϵSTORE^r across 10 evaluation runs. It is observable that profile data for the queries collected from the 2^{nd} set has substantially lower variance than that collected from the 1^{st} set, and, hence, the query profile data are more stable across runs.

The collected profile data breaks down the total query execution time ($\mathbf{T_{tot}}$) into the query parsing time ($\mathbf{t_{Pa}}$), query plan generation time ($\mathbf{t_{Pl}}$), and query plan execution time ($\mathbf{t_{Ex}}$). We compute the sum breakdown time ($\mathbf{t_{Bd}}$) from the profile data as t_{Bd}=t_{Pa}+ t_{Pl}+ t_{Ex}. All times are reported in milliseconds unless otherwise stated.

Table 6 shows the results for LDBC SNB. Column 1 shows the query abbreviation. Column 2 shows the scale factor of the dataset. Columns 3-5 show the results for Neo4j^s, Neo4j^i, and ϵSTORE^r respectively. We initially hypothesized that using reflection to retrieve field values may degrade performance due to excessive runtime type-checking. To test our hypothesis, we decided to implement field access using reflection (ϵSTORE^r) and the unsafe API (ϵSTORE^u). However, we observed no significant performance difference between these two modes of ϵSTORE. Hence, for brevity, we omit showing ϵSTORE^u results in Table 6.

For each system, we show the query parsing time (t_{Pa}), query planning time (t_{Pl}), query plan execution time (t_{Ex}), sum breakdown time (t_{Bd}), and the total query execution (T_{tot}). We use bold text for T_{tot}, and we use gray background to show the best value (smallest T_{tot}) in each row. OOM indicates out of memory (when the physical memory requirement exceeds ~64GB). The T_{tot} of queries in general scales with scale factor of the datasets with the exception of some queries (e.g., Q^A_{SNB},Q^B_{SNB}), that operate on node labels containing very

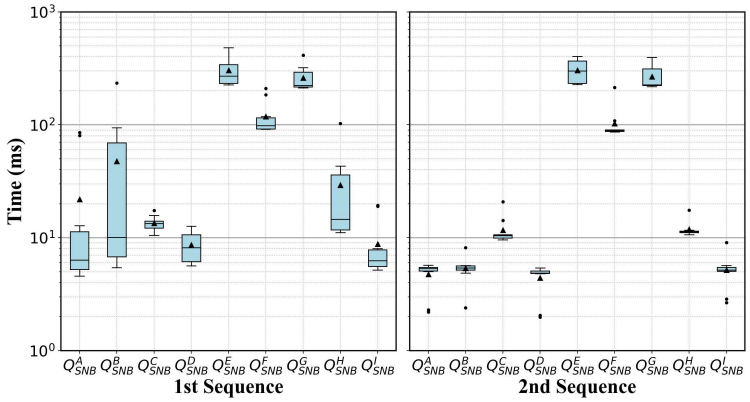
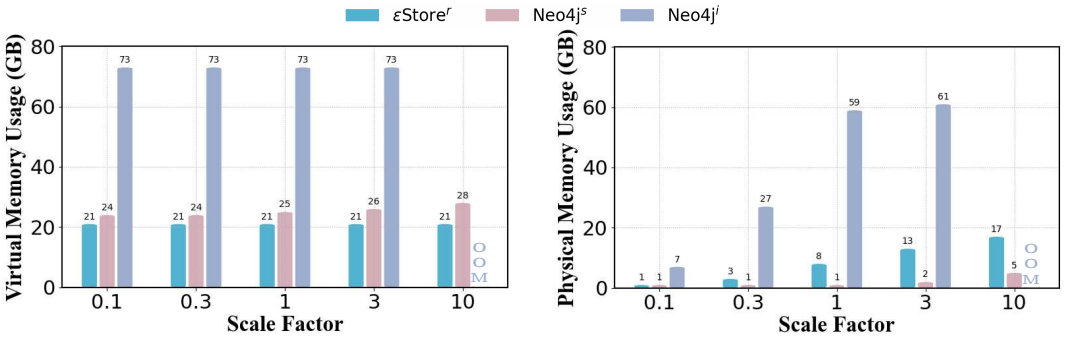


Figure 7 Query execution times for 1st and 2nd randomized sequence runs of ϵSTORE^r.



(a) SNB virtual memory usage in GB.

(b) SNB physical memory usage in GB.

Figure 8 Virtual and Physical memory usage in GB for LDBC SNB evaluation across scale factors. Query failures for exceeding memory are indicated by OOM.

few nodes. We can see that the T_{tot} of ϵSTORE is comparable or better than the production graph database for most of the queries. The graph database outperforms ϵSTORE for query Q^E_{SNB} because, Q^E_{SNB} matches referrer node with label containing the highest amount of nodes in the benchmark (20 million+ for SF 10). This shows that for testing purposes, where datasets are relatively smaller, ϵSTORE can be used as a lightweight graph store instead of a full fledged production graph database.

Memory usage. In addition to time, we also measured memory consumption for all the systems. We used pidstat [39] to collect memory consumption during evaluation runs. Figure 8 shows the peak memory usage for each system (one bar per system). The virtual and physical memory consumption in GB during running all the selected queries on the SNB for different scale factors is shown in Figure 8a and Figure 8b, respectively. We allow Neo4j^i and Neo4j^s to manage their own memory requirements [33], e.g., allocating page cache. We observe that the virtual memory usage of a system does not change significantly across scale factors of the SNB benchmarks. The physical memory usage, on the other hand, increases with increasing scale factors for all the systems. Neo4j^i requires the most physical and virtual memory and is OOM for the SNB scale factor 10 dataset. ϵSTORE^r consumes the least amount of virtual memory across the systems and is second only to Neo4j^s in terms of the least physical memory consumed. This is to be expected since ϵSTORE^r being in-memory,

Table 6 Total query execution time (T_{tot}) and its breakdown in milliseconds, for queries and datasets from the LDBC SNB benchmark. All reported times lower than 0.5 milliseconds are shown as 0. The maximum allocated physical memory for each evaluation run is 63GB.

Query	SF	Neo4j^s					Neo4j^i					ϵStore^r				
		t_{Pa}	t_{Pl}	t_{Ex}	t_{Bd}	T_{tot}	t_{Pa}	t_{Pl}	t_{Ex}	t_{Bd}	T_{tot}	t_{Pa}	t_{Pl}	t_{Ex}	t_{Bd}	T_{tot}
Q^A_{SNB}	0.1	11	29	0	40	**46**	1	33	3	37	**38**	0	0	0	0	**0**
	0.3	11	28	0	39	**44**	1	33	5	39	**40**	0	1	1	2	**3**
	1	10	24	0	34	**38**	0	31	24	55	**55**	0	0	1	1	**1**
	3	12	24	0	36	**40**	2	32	40	75	**76**	0	1	4	4	**5**
	10	11	23	0	35	**39**			OOM			0	0	4	5	**5**
Q^B_{SNB}	0.1	10	24	0	34	**39**	0	32	2	35	**36**	0	0	0	0	**0**
	0.3	12	27	0	39	**43**	0	33	5	38	**39**	0	0	1	1	**2**
	1	11	26	0	37	**41**	1	34	9	44	**45**	0	0	0	1	**1**
	3	10	23	0	33	**37**	0	33	52	86	**87**	0	1	4	5	**6**
	10	11	24	0	35	**39**			OOM			0	0	4	4	**4**
Q^C_{SNB}	0.1	11	25	0	36	**41**	0	32	2	34	**35**	0	0	1	1	**1**
	0.3	10	27	0	37	**41**	0	32	4	37	**37**	0	0	3	3	**4**
	1	10	24	0	34	**38**	0	30	18	49	**50**	0	0	3	3	**3**
	3	10	24	0	34	**38**	0	31	21	52	**53**	0	0	8	8	**9**
	10	11	25	0	36	**41**			OOM			0	0	19	19	**19**
Q^D_{SNB}	0.1	16	30	0	46	**51**	1	34	0	36	**36**	0	0	0	0	**1**
	0.3	15	30	0	45	**50**	1	34	0	35	**36**	0	1	2	2	**4**
	1	13	25	0	38	**43**	0	34	0	34	**35**	0	0	1	1	**1**
	3	16	26	0	43	**47**	3	34	0	37	**38**	0	1	2	3	**4**
	10	15	26	0	42	**46**			OOM			0	0	4	4	**5**
Q^E_{SNB}	0.1	10	23	0	33	**38**	1	30	0	31	**31**	0	0	7	7	**7**
	0.3	11	21	0	32	**36**	2	34	0	36	**37**	0	0	29	29	**30**
	1	11	22	0	34	**38**	4	29	0	33	**33**	0	0	114	115	**115**
	3	11	20	0	31	**36**	6	29	0	36	**37**	0	1	319	320	**321**
	10	11	20	0	31	**35**			OOM			0	0	866	866	**867**
Q^F_{SNB}	0.1	10	15	29	54	**58**	0	24	44	67	**68**	0	0	5	6	**6**
	0.3	10	16	66	92	**97**	1	25	107	133	**134**	0	1	15	16	**16**
	1	9	16	212	237	**241**	1	24	286	311	**312**	0	0	35	35	**35**
	3	8	16	530	554	**559**	0	23	754	777	**778**	0	0	87	87	**88**
	10	11	18	1531	1559	**1564**			OOM			0	0	258	258	**259**
Q^G_{SNB}	0.1	9	16	31	57	**61**	2	26	59	88	**88**	0	0	6	6	**7**
	0.3	9	16	102	128	**133**	0	24	206	230	**231**	0	0	30	31	**31**
	1	10	15	423	448	**453**	1	24	728	753	**754**	0	0	74	75	**75**
	3	9	15	1298	1322	**1326**	3	25	2484	2511	**2512**	0	0	244	244	**245**
	10	10	15	4393	4418	**4422**			OOM			0	0	915	915	**915**
Q^H_{SNB}	0.1	10	16	3	29	**33**	0	24	7	31	**32**	0	0	1	1	**1**
	0.3	10	16	6	33	**38**	0	24	16	40	**41**	0	1	4	4	**5**
	1	9	16	18	44	**48**	0	23	40	63	**63**	0	0	3	3	**3**
	3	9	16	45	70	**74**	0	23	105	128	**129**	0	1	10	11	**11**
	10	10	16	112	137	**143**			OOM			0	0	22	22	**22**
Q^I_{SNB}	0.1	9	17	2	28	**32**	0	24	3	27	**27**	0	0	0	0	**0**
	0.3	10	16	3	29	**34**	0	24	6	30	**31**	0	1	1	2	**2**
	1	9	16	8	33	**38**	0	24	14	38	**39**	0	0	1	1	**1**
	3	10	16	14	39	**43**	0	24	33	57	**57**	0	1	4	5	**5**
	10	9	15	40	64	**69**			OOM			0	0	6	6	**7**

stores all inserted data on RAM whereas Neo4j^s can store part of it on disk and can page it into RAM as and when required. We once again see that for smaller datasets, which is generally the case during testing, ϵStore's memory consumption is comparable or better than a production graph database.

In summary, the query execution times and memory consumption of ϵStore is on par or better than that of a production graph database for small datasets. Since, smaller datasets are typically the norm in testing environments, ϵStore provides an excellent light-weight alternative to a full fledged graph database for testing.

5.5 Data-structure Performance (RQ2)

We reuse 5 of the data structures from Thimmaiah et al. [57] that were used to evaluate OGO. We also introduce 4 additional data structures from the Eclipse Collections project in our evaluation.

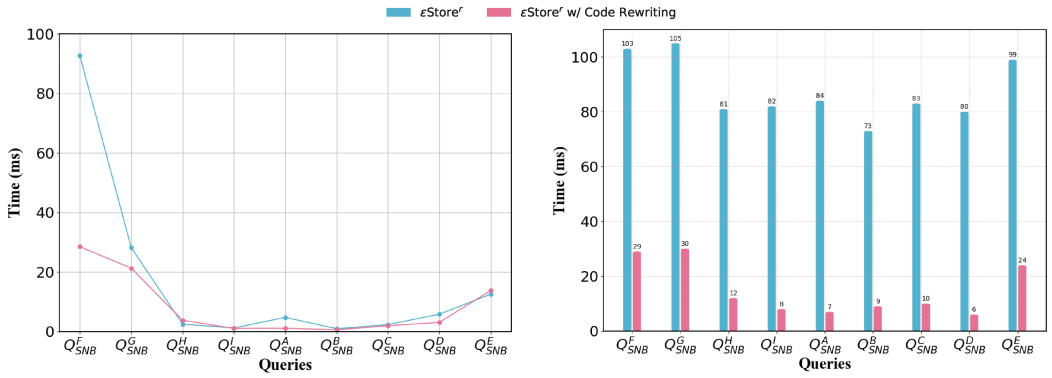
OGO supports two modes of operation, OGO^{Neo} and OGO^{Mem}. In both modes, the first step is to identify a subset of the heap that is relevant to the query. The 2 modes differ in the query engine used. OGO^{Neo} uses an external query engine (Neo4j) and OGO^{Mem} uses an in-memory query engine. We compared the 2 different modes of OGO with ϵStore by executing Cypher queries implementing imperative methods from data-structure libraries. We evaluated these queries for varying workloads (number of elements present inside the data structure). The results are shown in Table 7. We only report the total query execution time T_{tot} (in milliseconds) for each system. The data-structures are given in the first column.

Table 7 Total query execution time (T_{tot}) in milliseconds, for the *contains* or equivalent method reimplemented as Cypher query, on data structures for different modes of OGO and ϵStore. The time out (TO) duration used is 1 minute .

Data-structure	#Elements	OGO^{Neo}	OGO^{Mem}	ϵStore^{r}	ϵStore^{u}
JCF v17.0					
ArrayList	10^2	2231	264	89	81
	10^3	8169	486	97	90
	10^4	40261	888	133	116
	10^5	TO	TO	166	162
ArrayDeque	10^2	1632	399	88	86
	10^3	3377	497	105	89
	10^4	41301	1540	128	120
	10^5	TO	TO	156	166
HashMap	10^2	2391	315	78	77
	10^3	9479	516	88	100
	10^4	TO	2753	136	126
	10^5	TO	TO	356	394
LinkedList	10^2	24831	420	85	79
	10^3	TO	TO	83	88
	10^4	TO	TO	125	133
	10^5	TO	TO	232	216
Guava v32.1.3-jre					
ArrayTable	10^2	1711	358	82	76
	10^3	2742	415	94	90
	10^4	23298	1031	125	140
	10^5	TO	TO	237	228
Eclipse v11.1.0					
UnifiedSet	10^2	1592	514	78	76
	10^3	2916	595	87	90
	10^4	55464	3373	158	149
	10^5	TO	TO	223	239
UnifiedMap	10^2	1997	489	88	83
	10^3	5079	865	102	109
	10^4	TO	TO	111	113
	10^5	TO	TO	210	200
FastList	10^2	1637	453	80	77
	10^3	2750	540	89	87
	10^4	40243	1255	130	143
	10^5	TO	TO	172	161
ArrayStack	10^2	1820	411	78	82
	10^3	3538	741	87	84
	10^4	TO	2461	127	129
	10^5	TO	TO	153	153

The second column shows the workload, and finally, columns three through six give the total query execution times for OGO and ϵStore. We fixed the query execution time out (TO) duration to be 1 minute. We clearly see that both the modes of ϵStore consistently outperform those of OGO by at least an order of magnitude. Larger workloads for most of the considered data structures result in TO for OGO.

OGO^{Neo} is significantly slower than ϵStore because of using an external query engine which incurrs a heavy overhead due to repeated serialization and deserialization of the heap subset. This overhead increases for higher workloads due to increase in the heap subset size. ϵStore is also faster than OGO^{Mem} by an order of magnitude on average. This is primarily due to 3 factors. The first is that both the modes of OGO rely on tagging [45] of objects (assigning a `long` identifier) in the heap to identify those relevant to the query. Since other JVM processes such as the garbage collector (GC) also use tagging, both modes of



(a) Sequence. **(b)** Standalone.

Figure 9 Total query execution time (T_{tot}) for LDBC SNB queries in milliseconds. In (a), each query is executed in sequence inside a single JVM process; in (b), each query is executed in a separate JVM process.

OGO execute every query by first iterating through the entire heap and tagging every object to 0. This tag initialization time grows linearly with the number of objects in the heap. The second factor is, ϵStore is implemented purely in Java and thus benefits from Just-In-Time (JIT) compilation whereas OGO^{Mem} uses an in-memory query engine implemented in C++. Finally, the third factor is in the identification of the heap subset in OGO. Both the modes rely on the JVMTI `FollowReferences` [44] method to identify the heap subset. The `FollowReferences` method takes a user provided callback as one of its arguments and visits every object starting from JNI roots following its chain of references, reporting them to the callback. Providing class filters to `FollowReferences` only controls what objects are reported but not what objects are visited. This overhead of visiting objects not relevant to the query increases with increase in heap size. This is unlike ϵStore which stores references to the objects that might be queried.

We only compare total query execution time and not lines of code (LOC) since, both OGO and ϵStore use Cypher to implement the data-structure methods.

> In summary, ϵStore outperforms both the modes of OGO for Cypher queries re-implementing imperative methods from data-structure libraries. OGO is on-average an order of magnitude slower or worse. This is due to ϵStore benefitting from JIT compilation due to its pure Java implementation and storing references to all the queryable objects. OGO, on the other hand, incurrs heavy overhead due to its need to identify the heap subset relevant to the query for every query execution.

5.6 Imperative Code Rewriting Performance (RQ3)

We evaluate the performance of the code rewriting using LDBC SNB queries on the smallest scale factor dataset (0.1).

Figure 9 shows the evaluation results. In Figure 9a, we run each query sequentially in the same JVM process. (The order to run these queries is determined by the test runner, we observe the same trend with other orders). We observe that the code rewriting technique speeds up the first two queries; and for the subsequent queries the execution time is similar. There are two main reasons for the results. First, we use ANTLR to parse the query and the start up takes time [4], thus the first query of ϵStore^r takes much longer time than others. Second, after the warm up, JIT compilation kicks in and the difference becomes negligible.

We further evaluate the execution time when each query runs in a separate JVM process and the result is shown in Figure 9b. This result is consistent with our previous conclusion.

In summary, parsing and generation of query plans adds nearly an order of magnitude to the execution time. We can significantly speed up query execution by removing this overhead by injecting query plan execution code into the query call site at compile time.

6 Limitations and Future Work

We document potential future directions in this section.

Query optimization. Our current implementation of the query engine only generates the physical plan and directly executes it. In the future, we could introduce a logical plan and construct a physical plan from it. This could allow us to reason about query execution strategies at a more abstract level.

ID collisions. ϵSTORE stores references to captured objects using their class names and their IDs. The object's hash code is used as its ID. However, it is possible for two or more distinct objects to share the same hash code. Currently, ϵSTORE does not support retaining multiple objects with identical hash codes; if a newly captured object shares a hash code with an existing object, the existing reference is overwritten by the new one. Although such situations are theoretically possible, we did not encounter them during our evaluation.

Programming languages. We implemented ϵSTORE in Java due to our familiarity with the language. Modifying types of fields or number of fields in Java is hard. Class redefinition would require bytecode modification and loading in the class with a different classloader and then managing two different versions of the same class within a single JVM. This restricts the types of queries we can support in eStore (e.g., we cannot add new edges, we cannot add new node properties etc.). Other languages like Python or Smalltalk might allow eStore implementations to be more flexible and versatile. We leave design of an in-memory object graph store for other languages as future work.

Concurrency semantics. We defined semantics assuming sequential program execution. It would be interesting to define semantics for concurrent programs when an object might be accessed both inside and outside a store (or multiple stores simultaneously). However, that is outside the scope of the current work.

Query languages. ϵSTORE currently supports a subset of Cypher, which is the most popular query language. Future work could explore supporting other known graph query languages, e.g., GraphQL [19], Gremlin, SPARQL, and AQL [5]. Integration with languages that support both imperative and declarative traversals, such as Gremlin, could be especially well suited for ϵSTORE's data representation.

Cache layer for graph databases. The efficient in-memory graph store model of ϵSTORE makes it suitable to be used as a cache layer for persistent graph databases like Neo4j. The new programming style brought by ϵSTORE can further enrich the interoperability between the applications and the graph databases. We leave for future work the exploration of this direction.

7 Related Work

In this section, we cover the most closely related work, which we organize into: (1) graph databases storage, (2) language integrated queries, (3) object relational/graph mappers.

Graph databases storage. Graph databases [3, 60, 2, 9] are a type of NoSQL database. One of the most popular graph databases is Neo4j, which we discussed in this paper. Many other (proprietary) options are available including TigerGraph [58], Neptune [27], Nebula [37], JanusGraph [38], VelocityDB [36], Kùzu [16] and Memgraph [40]. VelocityDB is an in-memory object database integrated with C# and can be extended as a graph database, but it still introduces extra layer(s) of storage model abstraction and uses a specific set of APIs instead of a query language like Cypher.

Language integrated queries. [11, 42, 20, 54] Microsoft LINQ [42] is an integration of query capabilities directly into C# language. LINQ supports various data sources, including collections (e.g., List), SQL database, XML documents, and streams. Unlike LINQ, ϵStore is an in-memory graph backed object store. Including an object into ϵStore enables queries on it similar to those on data structures using LINQ. Apache Commons OGNL [18] is an open-source Expression Language (EL) for Java. It provides its own expression syntax to navigate and manipulate Java object graphs. However, it is not designed as a graph store, and does not provide the same level of expressiveness as graph query languages. OGO generalizes the idea behind LINQ's data structure queries and enables querying the entire Java heap. Unlike OGO that supports querying the state of the heap, ϵStore focuses on implementing an in-memory graph backed object store.

Object relational/graph mappers. Object-relational mapping (ORM) [59] is used to convert data between a (relational) database and the heap. In a way, object relational mapping techniques create an object database that can be directly manipulated within the program. Example of ORM include Hibernate [26]. There are also Object-graph mappers (OGM) for graph databases, such as Neomodel [15] and Renesca [14] for Neo4j. ϵStore is a graph backed object store and thus requires no additional mapping into memory objects.

8 Conclusions

We presented ϵStore, the first in-memory graph backed object store. ϵStore brings a programming paradigm shift, as it equates nodes in a graph with objects on the heap and relations among nodes with reference fields. It uses dynamic class generation and loading to create necessary schema (classes) to represent nodes and their properties. A subset of Cypher is used for querying the store, and each query returns a table of references. Additionally, ϵStore can transitively include an object already on the heap into a store, which enables complex queries for data and relations on already existing object graphs. Our evaluation shows the benefit of our approach. Besides being used as an object graph store, we expect that the combination of graph store features, object store features, implementation of a graph as an object graph, and ability to capture object graphs into a store will introduce new programming styles.

References

1 Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, David Püroja, Mirko Spasić, Benjamin A. Steer, Dávid Szakállas, Gábor Szárnyas, Jack Waudby, Mingxi Wu, and Yuchen Zhang. The ldbc social network benchmark, 2024. arXiv:2001.02299.

2 Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys*, pages 1–40, 2017.

3 Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, pages 1–39, 2008. doi:10.1145/1322432.1322433.

4 Antlr4. Save DFA for faster parser start up. Accessed September 6, 2024 from https://github.com/antlr/antlr4/issues/3682, 2024.

5 ArangoDB. Aql documentation. Accessed March 31, 2024 from https://docs.arangodb.com/3.12/aql/, 2024.

6 Andrea Arcuri, Man Zhang, Asma Belhadi, Bogdan Marculescu, Amid Golmohammadi, Juan Pablo Galeotti, and Susruthan Seran. Building an open-source system test generation tool: lessons learned and empirical analyses with evomaster. *Software Quality Journal*, pages 947–990, 2023. doi:10.1007/S11219-023-09620-W.

7 ASM. A Java bytecode engineering library. Accessed March 10, 2024 from https://asm.ow2.io/publications.html, 2024.

8 Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-tenant databases for software as a service: schema-mapping techniques. In *ACM SIGMOD International Conference on Management of Data*, pages 1195–1206, 2008. doi:10.1145/1376616.1376736.

9 Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *ACM Computing Surveys*, pages 1–40, 2023.

10 Alan Bundy and Lincoln Wallen. *Breadth-First Search*, pages 13–13. Springer Berlin Heidelberg, 1984.

11 James Cheney, Sam Lindley, and Philip Wadler. A practical theory of language-integrated query. In *International Conference on Functional Programming*, pages 403–416, 2013. doi:10.1145/2500365.2500586.

12 Shigeru Chiba. Load-time structural reflection in java. In *European Conference on Object-Oriented Programming*, pages 313–336, 2000. doi:10.1007/3-540-45102-1_16.

13 Oracle Corporation.. Linkedlist. Accessed March 30, 2024 from https://github.com/openjdk/jdk/blob/jdk-17%2B0/src/java.base/share/classes/java/util/LinkedList.java, 2019.

14 Felix Dietze, Johannes Karoff, André Calero Valdez, Martina Ziefle, Christoph Greven, and Ulrik Schroeder. An open-source object-graph-mapping framework for neo4j and scala: Renesca. In *Availability, Reliability, and Security in Information Systems*, pages 204–218, 2016. doi:10.1007/978-3-319-45507-5_14.

15 Robin Edwards. Neomodel product website. Accessed March 31, 2024 from https://neomodel.readthedocs.io/, 2024.

16 Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. Kùzu graph database management system. In *Conference on Innovative Data Systems Research*, volume 7, pages 25–35, 2023.

17 Eclipse Foundation. Eclipse collections. Accessed March 10, 2024 from https://github.com/eclipse/eclipse-collections, 2024.

18 The Apache Software Foundation. Apache ognl product website. Accessed March 31, 2024 from https://commons.apache.org/dormant/commons-ognl/, 2013.

19 The GraphQL Foundation. Graphql product website. Accessed March 30, 2024 from https://graphql.org/, 2024.

20 George Giorgidze, Torsten Grust, Alexander Ulrich, and Jeroen Weijers. Algebraic data types for language-integrated queries. In *Workshop on Data Driven Functional Programming*, pages 5–10, 2013. doi:10.1145/2429376.2429379.

21 Google. Guava: Google core libraries for Java. Accessed March 10, 2022 from https://github.com/google/guava, 2024.

22 Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. opencypher: New directions in property graph querying. In *International Conference on Extending Database Technology*, pages 520–523, 2018. doi:10.5441/002/EDBT.2018.62.

23 H2. H2 product website. Accessed February 25, 2024 from https://www.h2database.com/html/main.html, 2022.

24 Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, pages 287–317, 1983.

25 Graham Hamilton, Rick Cattell, and Maydene Fisher. *Jdbc Database Access with Java: A Tutorial and Annotated Reference*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1997.

26 Hibernate. Hibernate orm product website. Accessed March 31, 2024 from https://hibernate.org/orm/, 2024.

27 Amazon Web Services Inc. Amazon neptune product website. Accessed March 30, 2024 from https://aws.amazon.com/neptune/, 2024.

28 Neo4j Inc. Neo4j product website. Accessed November 10, 2022 from https://neo4j.com/, 2022.

29 Neo4j Inc. Values and types. Accessed March 10, 2024 from https://neo4j.com/docs/cypher-manual/current/values-and-types/, 2022.

30 Neo4j Inc. Bolt protocol documentation. Accessed February 25, 2024 from https://neo4j.com/docs/bolt/current/, 2024.

31 Neo4j Inc. Graphdatabaseservice. Accessed March 10, 2024 from https://github.com/neo4j/neo4j/blob/5.17/community/graphdb-api/src/main/java/org/neo4j/graphdb/GraphDatabaseService.java, 2024.

32 Neo4j Inc. Impermanentdbmsextension. Accessed March 10, 2024 from https://github.com/neo4j/neo4j/blob/5.17/community/community-it/it-test-support/src/main/java/org/neo4j/test/extension/ImpermanentDbmsExtension.java, 2024.

33 Neo4j Inc. Memory configuration. Accessed March 10, 2024 from https://neo4j.com/docs/operations-manual/current/performance/memory-configuration/, 2024.

34 Neo4j Inc. Neo4j-admin import. Accessed March 10, 2024 from https://neo4j.com/docs/operations-manual/current/tutorial/neo4j-admin-import/, 2024.

35 Neo4j Inc. Neo4j java driver. Accessed February 25, 2024 from https://github.com/neo4j/neo4j-java-driver, 2024.

36 VelocityDB Inc. Velocitydb product website. Accessed March 30, 2024 from https://velocitydb.com/, 2024.

37 Vesoft Inc. Nebula graph product website. Accessed March 30, 2024 from https://www.nebula-graph.io/, 2024.

38 JanusGraph. Janusgraph product website. Accessed March 30, 2024 from https://janusgraph.org/, 2024.

39 Canonical Ltd. pidstat - report statistics for linux tasks. Accessed March 10, 2024 from https://manpages.ubuntu.com/manpages/trusty/man1/pidstat.1.html, 2019.

40 Memgraph Ltd. Memgraph product website. Accessed March 30, 2024 from https://memgraph.com/, 2024.

41 Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: The Java unsafe API in the wild. In *Object-oriented Programming, Systems, Languages, and Applications*, pages 695–710, 2015. doi:10.1145/2814270.2814313.

42 Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: reconciling object, relations and xml in the. net framework. In *ACM SIGMOD International Conference on Management of Data*, pages 706–706, 2006.

43 Shamkant Navathe, Ramez Elmasri, and James Larson. Integrating user views in database design. *Computer*, 19(01):50–62, 1986. doi:10.1109/MC.1986.1663033.

44 Oracle. Java virtual machine tool interface (JVM TI) - followreferences. Accessed November 10, 2022 from https://docs.oracle.com/en/java/javase/17/docs/specs/jvmti.html#FollowReferences, 2022.

45 Oracle. Java virtual machine tool interface (JVM TI) - settag. Accessed November 10, 2022 from https://docs.oracle.com/en/java/javase/17/docs/specs/jvmti.html#SetTag, 2022.

46 Oracle. Method detail: identityhashcode. Accessed March 10, 2024 from https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/System.html#identityHashCode(java.lang.Object), 2023.

47 Oracle. Class (java platform se 17). Accessed March 10, 2024 from https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html, 2024.

48 Oracle. Classloader. Accessed March 10, 2024 from https://docs.oracle.com/javase/8/docs/api/java/lang/ClassLoader.html, 2024.

49 Oracle. Collections framework overview. Accessed March 10, 2024 from https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html, 2024.

50 Oracle. Field. Accessed March 10, 2024 from https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Field.html, 2024.

51 Oracle. Java interface resultset. Accessed Sep 12, 2024 from https://docs.oracle.com/en/java/javase/17/docs/api/java.sql/java/sql/ResultSet.html, 2024.

52 Oracle. Java jdbc api. Accessed February 25, 2024 from https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/, 2024.

53 Felipe Pontes, Rohit Gheyi, Sabrina Souto, Alessandro Garcia, and Márcio Ribeiro. Java reflection api: Revealing the dark side of the mirror. In *International Conference on the Foundations of Software Engineering*, pages 636–646, 2019. doi:10.1145/3338906.3338946.

54 Alex Potanin, James Noble, and Robert Biddle. Checking ownership and confinement. *Concurrency and Computation: Practice and Experience*, 16(7):671–687, 2004. doi:10.1002/CPE.799.

55 Philipp Seifer, Johannes Härtel, Martin Leinberger, Ralf Lämmel, and Steffen Staab. Empirical study on the usage of graph query languages in open source Java projects. In *International Conference on Software Language Engineering*, pages 152–166, 2019. doi:10.1145/3357766.3359541.

56 Gábor Szárnyas, Brad Bebee, Altan Birler, Alin Deutsch, George Fletcher, Henry A. Gabb, Denise Gosnell, Alastair Green, Zhihui Guo, Keith W. Hare, Jan Hidders, Alexandru Iosup, Atanas Kiryakov, Tomas Kovatchev, Xinsheng Li, Leonid Libkin, Heng Lin, Xiaojian Luo, Arnau Prat-Pérez, David Püroja, Shipeng Qi, Oskar van Rest, Benjamin A. Steer, Dávid Szakállas, Bing Tong, Jack Waudby, Mingxi Wu, Bin Yang, Wenyuan Yu, Chen Zhang, Jason Zhang, Yan Zhou, and Peter Boncz. The linked data benchmark council (ldbc): Driving competition and collaboration in the graph data management space. In *Technology Conference on Performance Evaluation and Benchmarking*, 2023.

57 Aditya Thimmaiah, Leonidas Lampropoulos, Christopher J Rossbach, and Milos Gligoric. Object graph programming. In *International Conference on Software Engineering*, pages 216–228, 2024.

58 TigerGraph. Tigergraph product website. Accessed March 30, 2024 from `https://www.tigergraph.com/`, 2024.

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

60 Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: a data provenance perspective. In *Annual Southeast regional conference*, pages 1–6, 2010.