# RuntimeSave: A Graph Database of Runtime Values

Matúš Sulír
Technical University of Košice
Košice, Slovakia
matus.sulir@tuke.sk

Antonia Bertolino
Gran Sasso Science Institute
L'Aquila, Italy
antonia.bertolino@gssi.it

Guglielmo De Angelis
IASI-CNR
Rome, Italy
guglielmo.deangelis@iasi.cnr.it

## Abstract

To persist variable values from running programs for development purposes, we currently recognize two strategies. Techniques based on examples are only useful to store small sample objects, while record-and-replay techniques are efficient but use opaque storage formats. We lack a middle ground offering acceptable scalability and easy queryability with standard tools. In this work-in-progress paper, we present RuntimeSave – a versatile approach to saving runtime values from the Java Virtual Machine (JVM) into a persistent Neo4j graph database. Its core idea is a two-layer graph model consisting of hashed and metadata nodes, inspired by Git internals. To reduce the written data volume, it packs certain object graph shapes into simpler ones and hashes them to provide partial deduplication. We also report a preliminary evaluation, applications, and future work ideas.

***CCS Concepts:*** • **Software and its engineering → Software maintenance tools**; *Object oriented languages*; Virtual machines; Integrated and visual development environments.

***Keywords:*** Java Virtual Machine, runtime values, graph database, debugging, persistence

**ACM Reference Format:**
Matúš Sulír, Antonia Bertolino, and Guglielmo De Angelis. 2025. RuntimeSave: A Graph Database of Runtime Values. In *Proceedings of the 17th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3759548.3763375

## 1 Introduction

Traditionally, source code is persistent, whereas runtime values are dynamic and ephemeral. A few approaches aim to bridge this gap. For instance, live programming [15] lets developers modify source code while runtime values remain intact. Record-and-replay tools and omniscient (back-in-time) debuggers persist the values for later execution reconstruction. However, these techniques treat the values as a means to an end, not as their primary focus. A notable exception is the family of approaches known as Example Centric Programming [7], Babylonian Programming [14], or Example Objects [12]. Here, runtime values called *examples* are either produced by manually written code snippets or collected semi-automatically from executions. Example mining [9] gathers runtime values from debugging sessions or tests and saves them for later use. Example objects have a variety of applications. For instance, source code lines can be visually augmented with sample variable values to enhance program comprehension [16]. Values can also be attached to a function as arguments to repeatedly execute it with the same input and enable live programming [14].

To store the collected runtime values, example-based tools often use structured text comments directly in the source code [11]. This does not scale to numerous or large example objects. Time-travelling debuggers and record-and-replay tools [3, 4, 8] tend to utilize custom binary formats, which are efficient in space and writing speed but difficult to query with existing declarative languages. Relational databases with SQL proved impractical for deep dynamic analysis [1]. A key-value database has been used but solely for objects' string representations, not complete structures [16].

Object Graph Programming [17] maps a Java program's runtime state to a Neo4j graph. However, it is used merely as temporary storage for querying with Neo4j's Cypher language, without persistence. Persistence raises issues such as metadata assignment, storage space limitations, and time efficiency in saving many similar objects, which our paper addresses. Object databases [2], e.g., ObjectStore [10], persist runtime values but are today limited to niche legacy deployments and have been superseded by object-relational and object-graph mapping (ORM/OGM). Object databases are tightly coupled to the host language and application, hindering ad-hoc querying and visualization with standard tools. They also lack content-based object deduplication, which is one of our goals. BOLD [13] models execution traces with variables as RDF (Resource Description Framework) triples, which are persistable in a database but less natural to query than Neo4j's property graph model.

In this paper, we present RuntimeSave: a middle ground between small-scale examples and large-scale execution recording. It serves as universal queryable persistent storage of runtime values collected from the JVM with associated metadata. By runtime values, we mean concrete variable values – either primitives or, more interestingly, complex objects. They

*VMIL '25, Singapore, Singapore*

come from various sources, ranging from manual curation of interesting objects in debugging sessions to automated sampling. Two methods improving scalability are described: packing simplifies graph shapes, and content-based hashing, inspired by Git, further reduces time and space requirements. We also briefly introduce potential applications, including querying, visualization, starting programs at any line, displaying sample values for comprehension, API exploration, unit test input generation, and runtime repository mining.

## 2 Storage Mechanism

RuntimeSave is currently designed as a debugger extension of an Integrated Development Environment (IDE). It collects data from a debugged application, creates an in-memory Java object graph from them, applies packing, hashing, and ID-hashing to minimize database writes, and saves necessary data into a Neo4j graph database.[1]

### 2.1 Data Collection

Since the runtime values are present in the debugged application's process, separate from the IDE, we need to transport them before further processing. For this, we use the Java Debug Interface (JDI). We distinguish three basic scenarios for the collection of values from the application being executed: manual object selection, savepoints, and sampling.

***Manual Object Selection.*** Developers often encounter objects of special interest. During debugging, it may be a set of objects from which a faulty state can be reconstructed. When designing a new API, it can be an exemplar case to demonstrate many features. To support this, we should allow the developer to manually select a given object in the object inspector (Variables view) while the program is suspended and save it with associated metadata, such as a descriptive name or the current date. Later, when a need to load the values arises (e.g., resuming a debugging session or viewing documentation related to this object), we could easily query the database for these objects based on the metadata.

***Savepoints.*** While manually saving objects would be useful, developers sometimes need an overview of multiple values occurring possibly at several code locations. For such cases, major IDEs provide logpoints.[2] Instead of suspending the program as breakpoints, a logpoint prints a message, usually showing the values of variables at the given line.

We extend this idea with *savepoints*. Instead of printing messages, savepoints store the values (full object graphs) of all local variables in scope, including `this`, into RuntimeSave's database. Like breakpoints and logpoints, savepoints can be customized with conditions, pass counts, caller filters, and other usual options.

***Sampling.*** For obtaining an overview of values occurring throughout the program, RuntimeSave offers a sampling mode. It suspends the program at every $n$-th executable line for the first $t$ executions of that line and collects the values of all local variables in scope. Both $n$ and $t$ are configurable. A particularly useful setting is $n = 1$ and $t = 1$, which we call an *all-lines collection mode*. Such data are suitable for visual augmentation of source code lines with sample variable values [16]. Setting $n = 1$ and $t = -1$ (infinity) creates a full execution recorder, though currently impractical for non-trivial programs due to performance limitations.

By default, we only collect data from the current project, excluding libraries and the standard Java API. Once collection from one line is complete, the data are processed in a separate thread, allowing the program to resume execution.

### 2.2 Graph Model

While reading the values via JDI, they are transformed into an in-memory graph representation consisting of plain Java objects that closely resemble the database model. Disregarding space and time constraints, such a graph could be written to the database using straightforward object-graph mapping.

Neo4j is a database based on the property graph model, which we briefly introduce using an address book example. A property graph consists of nodes, each with zero or more labels and properties. Labels, such as `:Person`, are analogous to classes in Object-Oriented (OO) languages. Properties are similar to attributes, e.g., `{name: "Ed", age: 1}`. Nodes are connected by directed edges called relationships, which have a type (such as `:HAS_ADDRESS`) and optionally properties, e.g, `{type: "home"}`. Relationships are analogous to references between objects in OO languages.

Now let us focus on RuntimeSave's graph model, whose sample instance is shown in Figure 1. It has two layers: *hashed nodes* (green, bottom) and optional *metadata nodes* (grey background, top). This is analogous to Git's internal storage mechanism [6] consisting of hashed objects (blobs, trees, and commits) and references, such as branches or tags, which point to these objects and act as metadata.

To simplify the object-graph mapping and algorithms on the in-memory graph, we introduced three restrictions for the hashed layer. First, each node has exactly one label (excluding hidden labels used for technical purposes). Second, outgoing relationships from a node with a given label have a fixed type and exactly one property, e.g., `Object` nodes may only have `HAS_FIELD` out-edges with a single `name` property. Third, edge property values must be unique among out-edges of a given node, e.g., a frame instance must not have duplicate variable names.

***Hashed Nodes Layer.*** The hashed nodes layer comprises *stack frames* and *values* of five basic types.

Stack frame nodes, labeled `Frame`, are created only in the savepoint and sampling collection modes. We model a stack frame simply as a set of local variables, each having its name stored in an outgoing relationship's property and its value

[1] https://neo4j.com/
[2] https://code.visualstudio.com/docs/debugtest/debugging#_logpoints

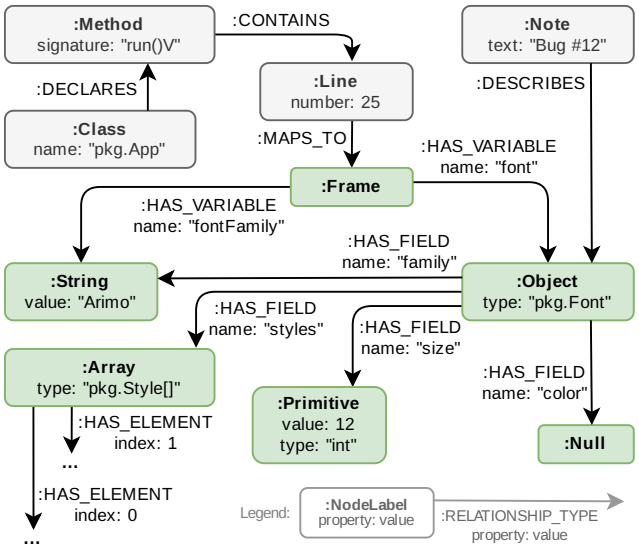


**Figure 1.** An instance of RuntimeSave's graph model

stored as its target node. Stack frames coming from instance methods include a special variable named "this".

Objects are modeled as nodes labeled `Object` with their fully qualified class name stored in the `type` property. Field names appear as properties of outgoing relationships, and their values as target nodes. Although strings are objects too, they have a special status in the JVM and JDI. We thus model them as `String` nodes, storing the value in a property instead of as separate characters, which also improves performance. An array is represented by an `Array` node with elements stored as target nodes. Primitive values, such as `int` or `double`, are modeled as `Primitive` nodes with a value and type stored in two properties. Null values are represented by a single shared `Null` node for the entire graph.

In addition to the properties displayed in Figure 1, every hashed node has two more properties: `hash`, described in Section 2.4, and `idHash`, explained in Section 2.5.

***Metadata Layer.*** The metadata layer comprises nodes that improve the ability to find the right value or stack frame node in the graph. Metadata nodes always point to the hashed nodes, not vice versa.

As shown in Figure 1, `Class` and `Method` nodes represent classes and their declared methods. A method can have source code lines denoted as `Line` nodes if the line number table is present in the class file. A line then maps to stack frames, so we can easily find frames coming from a specified line and use them to re-execute the line using stored variables, among other use cases. `Note` nodes are created manually by developers to mark objects related to issues or other relevant information, making them searchable later.

We envision many other kinds of metadata. For example, to better reconstruct past executions, we could record process identifiers, system times, JVM thread names, objects' unique IDs, and relative stack frame orders. For queries involving inheritance reasoning, we could store type hierarchies.

## 2.3 Packing

After constructing the in-memory graph, selected nodes from the hashed layer optionally pass through transformation algorithms that we call *packers*. These simplify the graph structure for two main reasons. First, runtime data volume is high, so we need to reduce the number of nodes and edges before storage. Second, packed graphs can be more comprehensible for humans and easier to query. Packing is reversible, so a graph can be unpacked without information loss.

One volume-reducing example is the Sparse Array Packer. In practice, many arrays contain mostly default values, such as 0 or null. We can replace the `Array` node with a `SparseArray` node, delete all the target nodes with default values, and store the array length explicitly in the `length` property.

A comprehension-focused example is the Linked List Packer. Java's `LinkedList` is implemented using nodes linked to previous and next items. While this low-level structure is suitable for certain analyses, we may prefer a more conceptual view: as an ordered, indexed collection. The packer transforms an `Object` of type `LinkedList` into a `LinkedListNode` with indexed out-edges directly referencing elements. Incidentally, this also reduces node and edge counts. The idea of comprehension-facilitating packers was inspired by the Moldable Debugger [12] and IntelliJ's Type Renderers, which display runtime values in more abstract, domain-specific ways. The key difference is that packers preserve enough implementation details to make the transformation reversible.

Another volume-reducing packer is the Same Primitives Packer. It leverages the fact that in JVM, primitive values cannot be aliased – no two fields or array elements can refer to the same primitive value. Before storing them in the database, the packer merges all primitives with the same value (e.g., all `int`s equal to 7) into a single node and redirects all original edges to it. During the unpacking, we unambiguously expand the merged node into multiple ones based on the number of its in-edges.

The Primitive Array Packer stores all values of a primitive array inside a single `elements` property of one `PrimitiveArray` node, thanks to Neo4j's homogeneous list type support. While this can make certain Cypher queries more challenging, it improves writing speed and space requirements.

Although we currently provide only four packers, the mechanism is extensible. It is possible to add a custom packer by implementing a simple four-method interface. Individual packers can be enabled or disabled as necessary, and their processing order is configurable.

## 2.4 Hashing

Packing partially reduces storage requirements, but the fundamental problem of duplication remains. Suppose we save

an array containing two different complex objects, each consisting of thousands of nodes and edges. Later, possibly after restarting the program, we save an array with the same elements, only swapped. Saving complete copies each time would be unsustainable. We therefore need an algorithm to identify which parts of the graph stored in the database correspond to the parts being saved, allowing us to store only the missing parts. This is known as the graph alignment problem, for which no exact polynomial-time algorithm exists.

Content-addressable systems using Merkle trees, such as Git, solve this problem elegantly and efficiently for trees [5]. Git is based on the premise that strong cryptographic hashing algorithms do not produce hash collisions in practice. Each file's content (i.e., "blob") is represented by its hash. A leaf directory's hash is computed from a sorted list of filenames paired with their blobs' hashes. This continues recursively: a parent directory's hash depends on the hashes of its children. Duplication is thus avoided simply by computing hashes and determining if they already exist in the repository.

RuntimeSave is also a content-addressable system but for graphs. Compared to general cases, our object graphs have specific characteristics: **i)** The graph being inserted into a database is directed, connected, node- and edge-labeled, and possibly cyclic. **ii)** Based on the restrictions we imposed in Section 2.2, each node's outgoing edges are unique and lexicographically ordered (e.g., an object's fields by name). This enables deterministic traversal and makes our approach practically feasible.

We have these requirements for our hashing algorithm: **i)** A hash is computed for every node, not only the entire graph. **ii)** We avoid encoding which node is a root because we want to detect also partial (sub-object) matches. **iii)** A hash represents the node's label and properties, the ordered property values of its outgoing edges, and – recursively – the same for all reachable nodes. In effect, it captures the full topology of the node's reachable subgraph. **iv)** No hash collisions occur in practice, so two nodes with the same hash represent JVM objects equal by deep value-based equality.

Neglecting reference equality for now (see Section 2.5), our scheme enables natural querying while achieving significant deduplication. This is illustrated in Figure 2 with nodes' hashes as uppercase letters, array indexes as numbered edges, and fields as edges with lowercase letters. Two arrays, hashed as A and G, both contain elements hashed B and C but in swapped order. If A is stored in the database, adding G does not duplicate nodes B to F. Only the node G and its direct out-edges are added, as its targets with hashes B and C (and thus their whole reachable subgraph) have already been stored.

Note that node A does not have knowledge of node G, and vice versa. In the resulting merged graph, the DFS (depth-first search) traversal from each node remains the same as in the original graphs. As a result, Cypher queries using only outgoing relationships remain natural and need not account for deduplication.

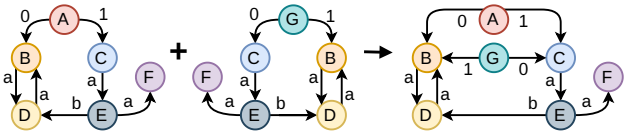


**Figure 2.** Hash-based deduplication example

General graph hashing algorithms are not well-suited to our case because they often ignore edge order or can produce the same hashes for non-isomorphic graphs. Therefore, we devised our own algorithm, whose naive version is shown in Algorithm 1. Procedure AssignHashes takes the set of all nodes in a graph as input and assigns a hash to each node independently of others. Unlike trees, graphs lack inherent hierarchy; using a spanning tree would yield different results depending on the starting node. The associative array *orders*, initialized on line 3, will map each node to its traversal order. The *hasher* (line 4) is a stream-like object to which we append unambiguous binary representations of data via the ≪ operator. In the end, it produces a hash (line 6). We use the SHA-224 function as it offers the shortest resulting hash among the algorithms available in standard Java distributions for which no collision has yet been found.

```
1  Procedure AssignHashes(nodes) is
2      foreach node in nodes do
3          orders ← empty Map
4          hasher ← empty SHA hasher
5          NodeHash(node, hasher)
6          node.hash ← hasher.produceHash()
7  Procedure NodeHash(node, orders, hasher) is
8      orders[node] ← orders.size
9      hasher ≪ node.label ≪ node.props ≪ START
10     foreach edge in node.outEdges do
11         hasher ≪ edge.property
12         if edge.target in orders then
13             hasher ≪ orders[edge.target]
14         else NodeHash(edge.target, orders, hasher)
15     hasher ≪ END
```

**Algorithm 1:** A naive version of the graph hashing

In procedure NodeHash, we first assign a traversal order to the given node (line 8). The node's label and properties are encoded into binary form and appended to the hasher on line 9. A special marker *START* denotes the beginning of the out-edge list to avoid ambiguity. We then iterate over the node's out-edges in their natural order (line 10). The property value of each edge is hashed (line 11). If the target node has already been visited, we encode it as its numeric traversal order on line 13. Otherwise, we continue recursively in a DFS manner (line 14).

Although the naive version has quadratic time complexity in the total number of nodes and edges, alternatives exist. We have implemented a variant that improves complexity for tree-shaped subgraphs and provides a constant-factor speedup by caching precomputed hashes for parts of data. Due to space constraints, we refer the reader to our source code, linked in Section 5.

### 2.5 ID-Hashing

Consider graph **F** in Figure 3, stored in a database. It contains three nodes with the same hash, B, which are value-equal but not reference-equal. Suppose only their hashes (B) were stored – the indices $B_1$ to $B_3$ appear solely for clarity. Now, we want to insert graph **G** into the database. Starting from the top, a node hashed D is not yet stored there, so we create it. Since enough B-hashed nodes already exist in the database, we aim to reuse them. However, we cannot arbitrarily match B-hashed nodes from **G** with those from **F**. For instance, connecting D's edge 0 to **F**'s $B_1$ and D's edge 1 to **F**'s $B_2$ would produce a non-tree graph instead of a tree. To find the correct attachment points, we would need to recursively scan the outgoing edges of multiple B-hashed nodes to find the matching graph topology. Such scanning would defeat the purpose of having hashes in the first place.

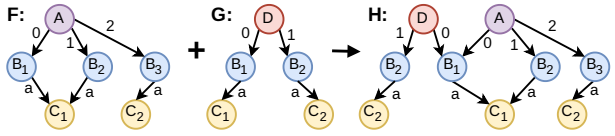


**Figure 3.** Deduplication considering reference equality

To solve this, we assign each node a special property called ID-hash, which encodes the topology of its reachable subgraph, considering not only value equality but also reference equality. First, we traverse a graph using DFS and assign each node its *copy* number computed as the number of nodes with the same *hash* encountered so far. Then, for each node $V$ in the graph independently, we construct a concatenation of tuples $(hash_{M1}, copy_{M1})$, ..., $(hash_{Mn}, copy_{Mn})$, where $M_1 ... M_n$ are nodes reachable from $V$ in DFS order, including $V$ itself. The ID-hash of $V$ is the SHA hash of this concatenation's binary representation.

Consider graph **F** in Figure 3. We mark all nodes with copy numbers shown as indices. After assigning ID-hashes, $B_1$ and $B_2$ encode two facts: they are distinct copies of nodes hashed B, and they both reference the same node $C_1$ in terms of reference equality. In graph **G**, $B_1$ encode its reference to $C_1$. Thus, the pair $B_1$–$C_1$ from **G** can be successfully reused and shared in the resulting graph **H**. However, there is no pair $B_2$–$C_2$ in **F**, so this branch will be created in the database and not reused in **H**.

Using DFS traversal order to assign copy numbers is only a heuristic: if we changed $B_2$ to $B_3$ in graph **G**, the branch could be reused since it already exists in graph **F**. Instead of traversal order, we could use JDI's unique object IDs or Java objects' identity hash codes. As these numbers are essentially random and reset between program runs, this would maximize reuse within a single program run but prevent reuse across multiple executions.

### 2.6 Saving Hashed Graphs to Database

After assigning each node its ID-hash, we save the graph into the Neo4j database, trying to minimize the query count and the amount of data sent. First, we detect strongly connected components (SCCs) in the graph being inserted. An SCC is a maximal subgraph in which each node is reachable from every other node. We construct the graph's condensation: a directed acyclic graph (DAG) created by contracting each SCC into a single node. To remove redundant edges, we compute the transitive reduction of the DAG, which is a minimal graph preserving the reachability relation between all node pairs. This results in a single-rooted acyclic graph expressing dependencies. Assuming no unfinished transactions, if a node with a given ID-hash exists in the database, then all nodes in its SCC and all reachable SCCs are present too.

The writing process leverages the ID-hash as a unique constraint – duplicate ID-hashes are forbidden. We start by sending a query trying to insert one node from the root SCC. Neo4j's reply indicates whether the node was really created or if a node with the same ID-hash already existed. If it existed, this means all its reachable nodes are stored in the database too, so we stop. If the node was created, we write the remaining nodes of that SCC and proceed to its target SCCs. We attempt to write one node from each target SCC. Neo4j returns a list of booleans indicating which nodes were created and which were already present. We stop the traversal of existing SCCs, write the remaining nodes of the newly created ones, and recursively continue this process until reaching terminal SCCs.

## 3 Preliminary Evaluation

For manual object selection and savepoints, performance is not critical unless handling large objects or placing savepoints at hotspot lines. We will focus our evaluation on the sampling mode, particularly the all-lines collection mode ($n = t = 1$). Running selected Apache Commons Lang unit tests, we observed a significant slowdown, about 30–40x. However, simply suspending the program with a breakpoint at every line, without collecting any data, caused roughly a 10x slowdown. We plan to replace JDI-based collection with bytecode instrumentation and a native JVM TI (Tool Interface) agent, expecting a radical speedup.

Thus, we decided to evaluate only the effect of packing and hashing, excluding JDI. We wrote a simple benchmark program that downloads an XML file, parses it using Java's DOM (Document Object Model) API, adds elements

**Table 1.** Comparison of database writing approaches

| Approach | Nodes | Edges | Time | Database size |
|---|---|---|---|---|
| plain | 26,304 | 41,306 | 2,343 ms | 8,904 kB |
| packed | 4,221 | 8,076 | 558 ms | 2,728 kB |
| hashed | 2,125 | 11,261 | 738 ms | 2,824 kB |
| pack & hash | 504 | 1,760 | 218 ms | 1,520 kB |

to it, and transforms it back to a string. After every line, all variables in scope were read using reflection and transformed to our in-memory graph representation, with a `Frame` node as a root. This resulting list of graphs was then written into the Neo4j database in four different ways: a plain approach without any processing, a packed-only graph, a hashed-only graph, and full processing (packing and hashing). We counted nodes and edges created in the database and measured the median processing and writing time, using 20 warmup rounds and 20 measurement iterations. The database was erased after each round. We also measured the size of the `data/databases/runtimesave` directory in Neo4j's home path. Based on the results in Table 1, packing and hashing significantly improved time and space requirements. Combining packing and hashing reduced the node count 52x, edge count 23x, time 11x, and database size 6x.

## 4 Applications

Runtime values stored by RuntimeSave could be applied in many ways; we highlight only a few.

***Querying and Visualization.*** Using the Neo4j graph database gives us access to its querying capabilities. For example, finding all objects with self-referencing fields is as simple as: `MATCH (n:Object)->(n) RETURN n`. On a database with about 100,000 nodes, such a query tends to complete in 20–30 ms, depending on the hardware.

Standard graph visualizations of runtime data aid comprehension. For instance, inspecting a Java DOM object graph, we promptly found out that parsed XML nodes are stored in a large array, not in a tree structure as we expected.

***Starting a Program at Any Line.*** To demonstrate RuntimeSave's potential, we developed an IDE extension that lets developers use any line (starting a Java statement) as an entry point. The method containing this line is executed using a combination of JDI and reflection, while bytecode instrumentation skips the execution of all earlier lines. For each variable in scope, we try to find a suitable value in the RuntimeSave's database. The currently implemented logic is simple, but we envision a graphical user interface where developers select objects from a list ordered by heuristics utilizing our metadata layer. Finally, the values are loaded from the database and assigned to the variables.

***Other Possible Applications.*** We could relatively easily implement a runtime diff tool that compares multiple stored objects, possibly from separate executions, field by field. To enhance program comprehension, an IDE could show expandable lists of sample variable values collected at each line. An API exploration tool would enable the execution of various methods on objects from the database, helping developers understand their behavior. Stored objects could also serve as actual or expected values in unit tests, especially when constructing them is difficult, such as in legacy software or with hard-to-reproduce errors. The database could be shared as a team knowledge base too. Finally, we envision an analogy to software repository mining, analyzing runtime information instead of static source code.

## 5 Future Work

RuntimeSave is implemented as an IntelliJ IDEA plugin, available at https://github.com/sulir/runtimesave. We have already implemented savepoints and sampling data collectors, four metadata node types, four packers, hashing with tree recognition and caching, ID-hashing, and database writing. Beyond the possible applications discussed earlier, we would like to extend RuntimeSave in several ways.

In the sampling data collector, we plan to replace JDI with bytecode instrumentation, a native C++ JVM TI Agent, and a custom binary serialization format to transfer the values from a debugged program to the IDE using shared memory. We expect a dramatic speedup from this. We would also like to further improve the hashing algorithm's efficiency, apply similar ideas to ID-hashing, and improve database access patterns. The effect of moving selected parts of RuntimeSave to high-speed cloud deployments could be explored too. If we achieved sufficient performance under certain configurations, data collection could potentially extend to simulated production environments.

In addition to the manual placement of savepoints by developers, IDEs could automatically suggest their locations and conditions. The current storage mechanism is not optimized for updating existing values, which is another future research direction. Data deletion could be handled similarly to garbage collection. Incompatibility of the saved objects with newer source code versions, although being a separate research problem, could also be explored. The scope of collected data in the hashed layer could be extended with non-final static variables. The metadata layer could be enriched with additional node types described in the paper.

RuntimeSave could also be ported to other language runtimes, such as Python. For its core features, it requires the ability to suspend and resume a debugged program, list variables in scope, and traverse the object graph.

## Acknowledgments

This work was supported by the Slovak Research and Development Agency under the Contract no. APVV-23-0408.

## References


[1] Mohammad R. Azadmanesh and Matthias Hauswirth. 2015. SQL for Deep Dynamic Analysis?. In *Proceedings of the 13th International Workshop on Dynamic Analysis (WODA 2015)*. ACM, New York, NY, USA, 2–7. doi:10.1145/2823363.2823365

[2] Francois Bancilhon. 1996. Object databases. *Comput. Surveys* 28, 1 (March 1996), 137–140. doi:10.1145/234313.234373

[3] Earl T. Barr and Mark Marron. 2014. Tardis: Affordable Time-Travel Debugging in Managed Runtimes. *ACM SIGPLAN Notices* 49, 10 (Oct. 2014), 67–82. doi:10.1145/2714064.2660209

[4] Jonathan Bell, Nikhil Sarda, and Gail Kaiser. 2013. Chronicler: Lightweight Recording to Reproduce Field Failures. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE, Piscataway, NJ, USA, 362–371. doi:10.1109/ICSE.2013.6606582

[5] Scott Chacon and Ben Straub. 2014. Git Internals – Git Objects. In *Pro Git* (2nd ed.). Apress, Berkeley, CA, Chapter 10.2, 358–365. https://git-scm.com/book/en/v2/Git-Internals-Git-Objects

[6] Scott Chacon and Ben Straub. 2014. Git Internals – Git References. In *Pro Git* (2nd ed.). Apress, Berkeley, CA, Chapter 10.3, 365–368. https://git-scm.com/book/en/v2/Git-Internals-Git-References

[7] Jonathan Edwards. 2004. Example Centric Programming. *ACM SIGPLAN Notices* 39, 12 (Dec. 2004), 84–91. doi:10.1145/1052883.1052894

[8] Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. 2013. Expositor: Scriptable Time-travel Debugging with First-class Traces. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE, Piscataway, NJ, USA, 352–361. doi:10.1109/ICSE.2013.6606581

[9] Eva Krebs, Patrick Rein, and Robert Hirschfeld. 2022. Example Mining: Assisting Example Creation to Enhance Code Comprehension. In *Proceedings of the Programming Experience 2022 (PX/22) Workshop*. ACM, New York, NY, USA, 60–66. doi:10.1145/3532512.3535226

[10] Charles Lamb, Gordon Landis, Jack Orenstein, and Dan Weinreb. 1991. The ObjectStore Database System. *Commun. ACM* 34, 10 (Oct. 1991), 50–63. doi:10.1145/125223.125244

[11] Fabio Niephaus, Patrick Rein, Jakob Edding, Jonas Hering, Bastian König, Kolya Opahle, Nico Scordialo, and Robert Hirschfeld. 2020. Example-Based Live Programming for Everyone: Building Language-Agnostic Tools for Live Programming with LSP and GraalVM. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2020)*. Association for Computing Machinery, New York, NY, USA, 1–17. doi:10.1145/3426428.3426919

[12] Oscar Marius Nierstrasz and Tudor Gîrba. 2024. Moldable Development Patterns. In *Proceedings of the 29th European Conference on Pattern Languages of Programs, People, and Practices (EuroPLoP '24)*. Association for Computing Machinery, New York, NY, USA, Article 18, 14 pages. doi:10.1145/3698322.3698327

[13] Dileep Kumar Pattipati, Rupesh Nasre, and Sreenivasa Kumar Puligundla. 2022. BOLD: An Ontology-Based Log Debugger for C Programs. *Automated Software Engineering* 29, 1 (May 2022), 43 pages. doi:10.1007/s10515-021-00308-8

[14] David Rauch, Patrick Rein, Stefan Ramson, Jens Lincke, and Robert Hirschfeld. 2019. Babylonian-style Programming: Design and Implementation of an Integration of Live Examples Into General-purpose Source Code. *The Art, Science, and Engineering of Programming* 3, 3 (2019), 9. doi:10.22152/programming-journal.org/2019/3/9

[15] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2018. Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness. *The Art, Science, and Engineering of Programming* 3, 1 (2018), 1:1–1:33. doi:10.22152/programming-journal.org/2019/3/1

[16] Matúš Sulír and Jaroslav Porubän. 2018. Augmenting Source Code Lines with Sample Variable Values. In *Proceedings of the 26th International Conference on Program Comprehension (ICPC '18)*. Association for Computing Machinery, New York, NY, USA, 344–347. doi:10.1145/3196321.3196364

[17] Aditya Thimmaiah, Leonidas Lampropoulos, Christopher Rossbach, and Milos Gligoric. 2024. Object Graph Programming. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 20, 13 pages. doi:10.1145/3597503.3623319