Modeling and Querying Versioned Source Code in RDF

Jacob Bellamy-McIntyre[1]

University of Auckland, Auckland, New Zealand,
`jbel071@aucklanduni.ac.nz`

Abstract. Source code management is an active and fundamental area of research where one of the key challenges is allowing developers to maintain an understanding of software projects when it is being actively developed in a distributed setting. Despite a number of well established practices and tools to help keep track of modifications to a project, there is a lack of a standard representation and query mechanism to integrate different repositories and serve fine-grained retrieval tasks. In this paper we propose modeling source code in resource description framework (RDF) triples as it is the main standard for sharing semantic information over the web. To support temporal queries over different source code versions we present a temporal extension of SPARQL that uses an in memory index of changes generated from a standard transaction log. We have built a prototype system to demonstrate that the approach is feasible, and present some preliminary results on query execution.

Keywords: temporal databases, RDF, SPARQL, static analysis

1 Introduction and Motivation

One of the key challenges in software engineering is supporting developers so they can understand and maintain software projects under active development. This problem is known as program comprehension and is especially challenging for developers new to a project. When reading source code it is natural for questions such as "which lines affect the value of this variable?", or "does this method call modify any data structure in addition to returning a value?". While different developer tools exist that aid in answering these kinds of questions, more expressive information needs cannot be resolved in a standardised way. This issue is aggravated when dealing with distributed source code that exists with multiple versions as they may apply to different systems and use different formats.

The Resource Description Framework (RDF) could be a suitable foundation for building a standardised source code model that allows for queries over these different formats using the SPARQL query language. As an increasing number of projects are hosted on public software repositories it makes sense to share information about those projects across the semantic web following linked data principles, and doing so gives access to a wide range of existing semantic web tools.

In this work we specifically wish to give support for temporal source code queries, where one can run their queries over the history of a project from its version control repository, and determine when different changes occurred to different source code artifacts. Being able to see the history of changes made to say, a class, can aid in program comprehension as it allows one to see the iterative process that went into building that class. Alternatively, one can use these queries to search for the introduction of bug patterns to a project in an automated fashion.

Our contributions are as follows: we propose a general RDF schema for source code based on abstract semantic graphs, we have developed a triplestore that maintains an index of all changes made to the store, we present the query language LSPARQL that allows for temporal queries using that index and which can be applied to typical triplestore transaction logs, and we have developed a prototype of our system focusing on Java source code to show that our approach can be practical. Our paper is structured as follows: section 2 covers the state of the art on querying source code and temporal RDF, section 3 discusses more in depth the problem we are trying to address, section 4 describes our approach in terms of our implementation, section 5 gives some preliminary results in terms of query execution speed on our prototype, section 6 details how we wish to further evaluate our approach in the future, and lastly section 7 summaries and concludes the work of this paper.

2 State of the Art

2.1 Querying Source Code

There have been numerous systems that have been developed to allow one to perform queries over source code artifacts. Early examples include OMEGA [1] and CIA [2] which used the relational model to represent the relationships between different source code artifacts, and allowed queries via SQL. Over the years several alternatives have been developed, such as the system ASTLog [3] which used an abstract syntax tree representation of source code with queries based around tree traversals, and CodeQuest [4] which was based on Datalog. Rather than use an underlying database to store a source code model, the JQuery eclipse plugin [5] instead answers queries directly against the eclipse API. More recently, the Wiggle system [6] implemented source code queries with Neo4J which uses the property graph model and the Cypher query language.

There have also been some implementations of modeling source code in RDF, for instance CodeOntology [7] and Evolizer [8]. In the case of CodeOntology there is no support for modeling multiple versions or supporting temporal queries over them. For Evolizer, while the authors do briefly describe how multiple versions of a project may be stored, there is no discussion on how one may form temporal queries or how they maintain the identity of individual artifacts across changes. The work of Ghezzi et al. [9] and Iqbal and Decker [10] allow higher level queries on software evolution by modeling the meta data of open source projects. EvoOnt [11] modeled entire source code repositories in RDF for the purpose of performing repository mining tasks such as detecting the introduction of code smells, but their model remains relatively coarse-grained and does not support queries on the statement level.

2.2 Temporal RDF

There has been some work on modeling time in RDF to be queried by SPARQL. The work of Guiterrez et al. [12] introduced Temporal RDF where every RDF triple additionally receives a temporal label describing the point in time or interval in which it is valid. They described how temporal labeling could be represented in standard RDF by performing reification on every triple.

While one could represent temporal triples in this way using any standard RDF triple store unfortunately it requires six additional triples for each triple that requires a temporal label describing an interval.

Besides being less space efficient than they could be, it also makes SPARQL queries cumbersome to write as they also need to refer to those six additional temporal triples per underlying triple in each query. The problem is compounded when when we want the matched triples to be valid over some common overlapping interval. This is because it would require comparing the valid intervals of each matched triple with all the other matched intervals. The cumbersomeness of the queries can be somewhat overcome by using a more concise language that translates temporal queries into regular SPARQL, as in [13], though efficiently answering these queries likely requires a specialised temporal index like tGrin [14].

There are a few alternatives to reification that have been explored, such as singleton properties where a unique predicate is used to denote a specific context to a relationship [15], and the work of Welty and Fikes [16] which instead represented fluents by giving a unique identifier to individuals depending on context. A separate approach again is to extend the basic RDF triple to also include an annotation which can describe its valid interval, such as is done with stRDF [17], aRDF [18] and AnQL [19]. The fundamental issue of this kind of solution is that these extended RDF triples are no longer standard RDF and so are not supported by standard tools, and are much less suited for sharing across the semantic web. Tappolet et al. [20] suggested using a separate named graph containing all the triples that hold for a specific interval and to use an index which specifies which named graphs apply for any particular time instant. This approach however is less applicable when few triples are valid for the same intervals.

Temporal queries also arise in the context of data archiving and versioning. The X-RDF-3x system [21] maintains an in memory index for triples that also includes when each triple was added or deleted, which they use for single version 'time travel' queries. R&WBase [22] similarly supports single version queries, but also takes into account standard version control operations like branching and merging to determine whether a triple holds for a particular version. The BEAR test suite [23] evaluates different archiving strategies over different stores using temporal queries written in AnQL.

3 Problem Statement and Contributions

The goal of our research was to create a system that can parse source code from a repository into RDF which would allow us to use source code queries with SPARQL. The challenge was designing our system in such a way that we could run queries to find changes made to specific source code artifacts in the repository, and to allow queries over any historic state without needing to materialize past versions. This presents three key problems:

1. Source code projects can span hundreds of thousands of lines. If we directly represent the abstract syntax tree of each class in RDF, this can easily become tens of millions of artifacts.

2. We require a means of supporting efficient temporal queries. Temporal reification would require a large number of additional triples. Extending triple patterns into quadruples with timestamps means that our triples would not be consumable by standard RDF triplestores.
3. When presented with two separate versions of the same source code, we require a way to associate artifacts in one version to the other. If we do not, then practically we cannot track changes to artifacts.

The major contributions of the PhD are a new schema for modeling source code in RDF and an extension to the SPARQL query language that allows for temporal queries over a triplestores transaction log. We have additionally created a proof of concept prototype system for Java source code.

4 Research Methodology and Approach

4.1 Overview

Our system has three major components. The first major component is our parser, which takes abstract syntax trees generated by a public library and uses those to construct an abstract semantic graph represented in RDF triples. The second is the underlying triplestore which we have developed ourselves called PDStore which uses a hash based indexing scheme. A key feature of PDStore is that by default it indexes all changes that are made in the transaction log so that queries can be run against any historic state. The query language LSPARQL is the last major component, and it allows one to specify temporal queries which are run against the changes described in the transaction log.

We parse all the classes of that project into abstract syntax trees, which our parser then uses to generate an abstract semantic graph represented as a set of RDF triples. Once all the classes have been parsed, we commit all the RDF triples in a single transaction which adds them to the log and assigns each the timestamp of the commit. We then sequentially repeat this process for each subsequent version we wish to run queries over. When generating a new abstract semantic graph after the first, we have to compare with the previous version to identify triples that need to be removed as they no longer exist in the current version. Artifacts which are unchanged naturally gain no new triples. Once all versions have been parsed they can be queried using LSPARQL. While currently our implementation focuses on Java, this process can be straight forwardly repeated for other declarative programming languages.

4.2 Abstract Semantic Graph Representation

An abstract syntax tree is a common representation of source code that is based on the parse tree used by compilers that prunes away much of the low level parsing such as punctuation. An abstract semantic graph (ASG) is a further abstraction that treats the program as one expression, and each vertice is a subexpression. Abstract semantic graphs differ from abstract syntax trees in that there can be additional edges to different vertices (for instance denoting invocation) and may contain back edges such as with recursion. Furthermore, duplicate subexpressions that occur in different portions of the

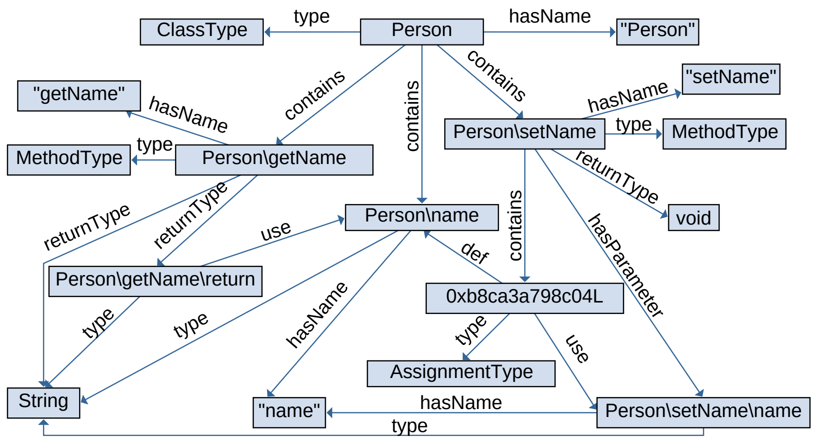

Fig. 1. ASG of a Simple Class

program can reuse the same vertices. In our prototype system we currently parse all Java artifacts down to the statement level. Figure 1 shows an example of an ASG we create, with simplified identifiers.

We assume the abstract syntax tree for the source code is available as they are a standard feature of source code parsing libraries. Given an abstract syntax tree, generating an abstract semantic graph is not particularly difficult. It just requires performing name resolution to associate the usage of some identifier with its definition. Common subexpressions could be identified by using a hashing scheme like what is done with merkle trees. What is much more difficult is determining the changes made to specific artifacts across subsequent graphs.

Analogous to the notion of a superkey in the relational model, different artifacts can have different properties which we can use to define a unique identity for that artifact, and if two artifacts across two different versions have these same properties then they are the same artifact. The most obvious of these are identifiers, such as class names, method names, and variable names. If two classes in separate versions have the same fully qualified name including the package, we can be fairly certain that they are the same artifact regardless of whichever changes have been made. Likewise we can do the same with methods if we include the method signature and containing class, and for variable declarations if we include the scope of the variable. These same names can straightforwardly be used in defining the IRIs used in RDF to identify them. For example, the bar() method in the foo class could be recorded as “projectURI:foo/bar()”. The deficiency of doing this is that if an artifact is renamed it assumes a new identity. If the rename is of the class, then this will also change the identity of all containing methods and variables.

For artifacts that cannot be uniquely identified by a name we see two viable options. The first is to use a combination of type and position. For example "the first while loop in the bar() method in the Foo class". The second would be to define its identity by all of its properties and sub properties, such as "the while loop with this condition and these statements". The former approach is a straight forward way of tracking the identity of artifacts without identifiers for which we can still create meaningful IRIs, but its identity can be usurped by the insertion of another artifact of the same type before it and its identity will not persist through a rename of a parent artifact or if the artifact is moved. The latter approach essentially would give us hash based identifiers that are immutable and so the only change of them that we can query is which artifacts contain them. They are perfectly suited for representing common subexpressions though, and can be recycled for representing reused code fragments across a project. While there is some flexibility as to which identity scheme to use, the decision we made with respect to this implementation which we used for our evaluation was to use type and positional identification for block constructs like for loops and while loops as well as variable assignments, and use hash based identifiers for other statements and expressions. In our prototype, we have used string based IRIs for representing the mutable artifacts, and hash-based integer identifiers for the immutable expressions.

4.3 PDStore

PDStore is an RDF triplestore that uses a hash based index for all triples added and deleted from the store. The standard (s, p, o) triple is instead represented with a quintuple (t, c, s, p, o) where t is a timestamp and c is a change type that is either an add change $+$ or a removal change $-$. The modification of a triple is represented by the deletion of the original triple, and the addition of a new triple. This representation corresponds to the minimum amount of information one would expect from a standard triplestore's transaction log. Internally String literal values such as URI identifiers are represented with 128 bit integers, with the strings mapped to them using a dictionary.

Indexing is performed using a pair of hash based indexes called the InstanceInstance index, and the RoleInstance index. The InstanceInstance index is a hashmap that takes as a key both a subject and object pair, and has as its value a list of all triples added and deleted that used both that subject and object in temporal order. As such, it is used directly to answer $(?t, ?c, s, ?p, o)$ queries. The RoleInstance index takes the predicate as a key, and has a second hashmap as its value. The second hashmap takes either a subject or object as a key, and once again returns a list of all corresponding triples that were added or deleted in temporal order. As such, it answers directly both $(?t, ?c, ?s, p, o)$ queries and $(?t, ?c, s, p, ?o)$ queries [1]. Table 1 describes the runtime complexity of the different query patterns over our log. Of note are queries where the change type is denoted by e which is used for queries on a single snapshot. For all patterns the runtime complexity for $+$, $-$, and $+-$ (wildcard) change types is unchanged, so we only list those for $+$. As in our prototype system there is a small finite number of roles, we assume the time it takes to iterate over all roles to take $O(1)$ time.

[1] When a predicate/object pair is added to the roleInstance index the predicate provides a slightly different key than if the object was given as a subject

Table 1. Index Query Efficiency

Change Patterns	Iterator retrieval	Get next
All queries with variable timestamp	$O(1)$	$O(1)$
$(t, +, s, p, o)$, $(t, +, s, ?p, o)$, $(t, +, ?s, p, o)$, $(t, +, s, p, ?o)$	$O(\log n)$	$O(1)$
$(t, +, ?s, ?p, ?o)$, $(t, +, s, ?p, ?o)$, $(t, +, ?s, ?p, o)$	$O(n)$	$O(n)$
$(t, e, s, ?p, o)$, $(t, e, ?s, p, o)$, $(t, e, s, p, ?o)$	$O(\log n)$	$O(n)$
(t, e, s, p, o), $(t, e, ?s, ?p, ?o)$, $(t, e, ?s, ?p, o)$, $(t, e, s, ?p, ?o)$	$O(n)$	$O(n)$

The $O(n)$ complexity can arise for those patterns having to visit multiple buckets in the hashmap. It might be that the earliest change in a bucket occurs after the specified timestamp and so would need to be skipped. In the worst case, the number of buckets which would have to be checked and then skipped is proportional to n. For a case like $(t, e, s, ?p, o)$ only a single bucket needs to be visited, but it is possible that a triple might be added and removed repeatedly but only a single one of those changes is related to the specified timestamp. Practically however, the expected complexity $O(\log n)$ in normal use cases.

4.4 LSPARQL

As mentioned in the previous subsection, PDStore uses a quintuple representation of triples of the form (t, c, s, p, o), where t is a timestamp, and c is a change type. For LSPARQL queries, the valid change types are $+$, to denote an added triple, $-$ to denote a deleted triple, e which means 'exists' and which is used for queries on a specific snapshot. Regular SPARQL queries which do not specify a timestamp and change type run the query on the most recent state, and are equivalent to a query that uses the most recent timestamp with a change type of e and thus give the same results as it would on a conventional triplestore that just had the latest snapshot. Currently LSPARQL is limited to conjunctive queries with FILTER.

The temporal variable $?t$ is assigned a single timestamp denoting a point in time in the case of patterns with change types $+$, $-$, and $+-$, and a pair of timestamps denoting a valid interval in the case of e patterns. Computing the valid interval for a given change takes $O(1)$ time, as we store a reference from an added change to one that removes it, and vice versa. Performing a temporal join with two single timepoints requires them to be the same time point. When performing a temporal join on two intervals however, we get the intersection of those intervals. For instance, given the triple $(Bob, likes, Alice)$ valid in the interval $(1, 4)$, and $(Alice, likes, Bob)$ valid in the interval $(2, 5)$, the query $(?t, e, ?x, likes, ?y)$, $(?t, e, ?y, likes, ?x)$ then the corresponding mapping μ would give $\mu(?t) = (2, 4)$. Sometimes a temporal join must be performed between a single time point and an interval, as would be the case in a query like $(?t, +, ?x, likes, ?y)$, $(?t, e, ?y, likes, ?x)$. In which case the mapping μ for this query would give $\mu(?x) = Alice$, $\mu(?y) = Bob$, and $\mu(?t) = 2$.

We support the standard SPARQL filters, and additionally have incorporated filters for the different temporal relations described in Alan's Interval Algebra, such as meets, overlaps, and during.

Table 2. Query Evaluation

Query #	Time(seconds)	Query Description
1	0.16	All recursive methods in the latest version
2	0.9	All classes in the current version which previously removed a method
3	0.11	All classes which implemented an interface that had some methods that were later removed
4	3.97	All methods whose return type changed at some point
5	0.02	All variables that had at some point been both incremented and decremented
6	0.09	SwitchTypes that later added a new SwitchCase
7	2.02	All pairs of assignments that modified the same variable at different points in time
8	0.007	All classes that existed when the 'HashQuery' class was added

5 Current Results

To demonstrate that our approach is viable, we have opted to parse the Chronicle Map project, a Java based key-value store publicly available on Github[2]. The latest version of the project consists of 381 Java files over 57603 lines of code. While the project itself is only modestly large, the abstract semantic graph model we are using is fairly fine grained, and we are also wishing to account for every historical version which was developed over 2330 commits. In total our in memory indexes account for around nine million separate triples.

We present our proof of concept by showing that our query system can answer some ad hoc temporal queries in a reasonable amount of time. The machine running our implementation has an intel I7-2600 cpu and 12 gigabytes of RAM. For each temporal query we test the time taken to execute the query, and then iterate over all of the results. The recorded time for each query is the average over ten separate runs. The total in memory usage was 3.7 gigabytes. The results we can see in Table 2 demonstrates our proof of concept, as all queries were answered within a few seconds.

6 Evaluation Plan

Our current evaluation is very limited as it is simply a proof of concept. A proper evaluation would consist of evaluating several different metrics. Firstly, a performance evaluation should compare our query evaluation with other implementations. For other source code query systems we should compare the speed of query execution using a single version, using a much larger set of queries, and with projects of varying size. To evaluate our temporal queries, we can potentially compare it to different repository mining tools to see if we can generate a comparable amount of information about the repository in

[2] https://github.com/OpenHFT/Chronicle-Map

a faster amount of time, or we could compare against temporal databases using more general temporal datasets.

Secondly, we need to consider expressibility. We need to consider other temporal SPARQL implementations, and determine whether LSPARQL can formulate equivalent queries by way of a formal comparison of our semantics. For our source code model, we should try to establish which patterns of changes we can easily capture and those we cannot.

The last main metric we would want to evaluate our system is usability. We would like to do a study where we provide some software developers a sample project repository, and we ask them to perform some tasks such as bug and feature location, as well as determine when and by whom those features or bugs were implemented. We would then introduce them to our query language, teach them how to write our queries, and ask them to perform the same tasks using our query language. We would record the time to do these tasks with both approaches, and give a questionaire asking whether they found using our queries easier or harder, and whether they could practically see themselves ever wanting to use it.

7 Conclusion

In this paper we have presented our proposal for using RDF for modeling versioned source code. In section 4.1 we described how we can accomplish this using an abstract semantic graph representation. Incorporating temporal information into RDF triple-stores is an open research problem, and in section 4.3 we described a simple hash based temporal index based on entries in a standard transaction log. Then, in section 4.4 we proposed the language LSPARQL, a natural temporal extension of SPARQL that can use such a temporal index to answer temporal queries over transaction time. In section 5 we described a prototype system we have developed that parses Java source code into RDF and which we query with LSPARQL, and have provided some preliminary results. In section 6 we describe how in the near future we wish to improve the evaluation.

References

1. Linton, M.A.: Implementing relational views of programs. In: ACM SIGSOFT Software Engineering Notes. vol. 9, pp. 132–140. ACM (1984)
2. Chen, Y.F., Nishimoto, M.Y., Ramamoorthy, C.: The c information abstraction system. IEEE Transactions on software Engineering (3), 325–334 (1990)
3. Crew, R.F., et al.: Astlog: A language for examining abstract syntax trees. In: DSL. vol. 97, pp. 18–18 (1997)
4. Hajiyev, E., Verbaere, M., De Moor, O.: Codequest: Scalable source code queries with datalog. In: ECOOP. vol. 6, pp. 2–27. Springer (2006)
5. Janzen, D., De Volder, K.: Navigating and querying code without getting lost. In: Proceedings of the 2nd international conference on Aspect-oriented software development. pp. 178–187. ACM (2003)
6. Urma, R.G., Mycroft, A.: Source-code queries with graph databases-with application to programming language usage and evolution. Science of Computer Programming **97**, 127–134 (2015)

7. Atzeni, M., Atzori, M.: Codeontology: Rdf-ization of source code. In: International Semantic Web Conference. pp. 20–28. Springer (2017)
8. Würsch, M., Ghezzi, G., Reif, G., Gall, H.C.: Supporting developers with natural language queries. In: Software Engineering, 2010 ACM/IEEE 32nd International Conference on. vol. 1, pp. 165–174. IEEE (2010)
9. Ghezzi, G., Würsch, M., Giger, E., Gall, H.C.: An architectural blueprint for a pluggable version control system for software (evolution) analysis. In: Proceedings of the Second International Workshop on Developing Tools as Plug-Ins. pp. 13–18. IEEE Press (2012)
10. Iqbal, A., Decker, S.: Integrating open source software repositories on the web through linked data. In: Information Reuse and Integration (IRI), 2015 IEEE International Conference on. pp. 114–121. IEEE (2015)
11. Tappolet, J., Kiefer, C., Bernstein, A.: Semantic web enabled software analysis. Web Semantics: Science, Services and Agents on the World Wide Web **8**(2-3), 225–240 (2010)
12. Gutierrez, C., Hurtado, C.A., Vaisman, A.: Introducing time into rdf. IEEE Transactions on Knowledge and Data Engineering **19**(2) (2007)
13. Perry, M., Jain, P., Sheth, A.P.: Sparql-st: Extending sparql to support spatiotemporal queries. In: Geospatial semantics and the semantic web, pp. 61–86. Springer (2011)
14. Pugliese, A., Udrea, O., Subrahmanian, V.: Scaling rdf with time. In: Proceedings of the 17th international conference on World Wide Web. pp. 605–614. ACM (2008)
15. Nguyen, V., Bodenreider, O., Sheth, A.: Don't like rdf reification?: making statements about statements using singleton property. In: Proceedings of the 23rd international conference on World wide web. pp. 759–770. ACM (2014)
16. Welty, C., Fikes, R., Makarios, S.: A reusable ontology for fluents in owl. In: FOIS. vol. 150, pp. 226–236 (2006)
17. Koubarakis, M., Kyzirakos, K.: Modeling and querying metadata in the semantic sensor web: The model strdf and the query language stsparql. In: Extended Semantic Web Conference. pp. 425–439. Springer (2010)
18. Udrea, O., Recupero, D.R., Subrahmanian, V.: Annotated rdf. ACM Transactions on Computational Logic (TOCL) **11**(2), 10 (2010)
19. Lopes, N., Polleres, A., Straccia, U., Zimmermann, A.: Anql: Sparqling up annotated rdfs. In: International Semantic Web Conference. pp. 518–533. Springer (2010)
20. Tappolet, J., Bernstein, A.: Applied temporal rdf: Efficient temporal querying of rdf data with sparql. In: European Semantic Web Conference. pp. 308–322. Springer (2009)
21. Neumann, T., Weikum, G.: x-rdf-3x: fast querying, high update rates, and consistency for rdf databases. Proceedings of the VLDB Endowment **3**(1-2), 256–263 (2010)
22. Vander Sande, M., Colpaert, P., Verborgh, R., Coppens, S., Mannens, E., Van de Walle, R.: R&wbase: git for triples. In: LDOW (2013)
23. Fernández, J.D., Umbrich, J., Polleres, A., Knuth, M.: Evaluating query and storage strategies for rdf archives. In: Proceedings of the 12th International Conference on Semantic Systems. pp. 41–48. ACM (2016)