Lessons Learned from Analyzing Requirements Traceability using a Graph Database

Maksim Goman, Michael Rath, Patrick Mäder
Software Engineering for Safety-Critical Systems
Technische Universität Ilmenau
{maksim.goman, michael.rath, patrick.maeder}@tu-ilmenau.de

Abstract

Established traceability among development artifacts allows to apply structured analysis in order to answer questions posed by stakeholders. Typically, the artifacts and their links are stored in relational databases. However, answering trace related questions involves finding paths and patterns in the artifact graph - a difficult task to perform using generic query languages. Mapping the artifact and link data onto graph databases and utilizing specialized query languages may overcome this limitation. In this paper, this mapping from a relational traceability dataset to a graph database is demonstrated. Afterwards, the advantages and disadvantages of the approach are investigated by calculating three trace metrics, heavily relying on graph patterns, using a graph query language. Overall, utilizing a graph database proved to simplify traceability analysis.

1 Introduction

The development of software and systems creates a manifold of artifacts, such as requirements specifications, design diagrams, source code and tests. Furthermore, relations among the problems artifacts exists, established as links, e. g. from design specifications to their implementation in source code. The artifact and link information allows to apply traceability analysis, such as coverage analysis, and requirements validation. However, to be effectively used, the collected trace information needs to be easy to query and process [7]. Most tools used in software and systems development use relational database to store artifacts and their links. While relational databases have matured query languages for data access and manipulation, they are not so convenient for traceability analysis. Especially, tracing between artifacts, which is usually modeled as multiple joins, becomes complicated as the number of traversed artifacts increases [1]. This kind of databases were not designed for processing of such graph-like data. Creating software for the research is an effective way, but it seems not efficient because it requires much programming expertise and change management. Thus, recently researchers started to investigate alternative data storage solutions, such as NoSQL databases [3].

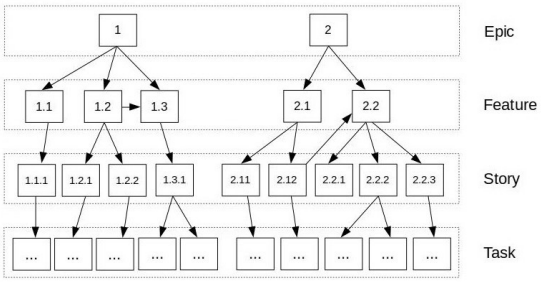

Figure 1: Illustration of a requirements traceability graph after decomposition for agile software development.

2 Sample Problem for Data Analysis with a NoSQL Database

The overall goal of our study was to calculate traceability metrics described by Rempel et. al [6] using a graph database. Requirements decomposition reveal dependencies between traceability artifacts and represents a traceability graph (see Figure 1). There, the authors proposed three metrics to assess and quantify the complexity of relationships between requirements: (1) number of related requirement (NRR), (2) average distance to related requirements (ADRR), and (3) requirement information flow (RIF). A combination of requirements traceability metrics was used to predict requirements defects (bugs) within a project. Every metric deals with links among artifacts resulting in path search. These paths can span many intermediate artifacts and also form loops. As data source, we used the recently published "The IlmSeven Dataset" [5], which contains artifacts from seven large open-source projects mined from issue tracking systems (ITS) and version control systems (VCS). The typed artifacts from ITS, called *issues*, describe features, improvements, and defects of the system. The VCS provides artifacts in the form of change sets, called *commits*, which bundle modifications of (source code) files. Furthermore, the dataset contains traces among the issues, as well as ones from issues to source code, e. g.

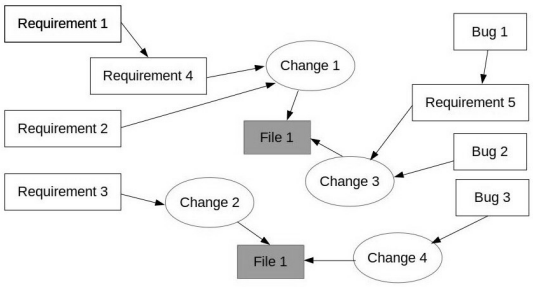

Figure 2: A requirement is related to a bug, if both artifacts can be traced to the same source code files.

from improvements to their implementation.

The description model proposed in [6] considers a rather ideal structure of agile requirements with "epic", "feature", "story" and "task" as issue types. This model is not straightly applicable for the dataset used in this study. In fact, the projects in the used dataset [5] contain a considerable number of requirements that are not traced to other requirements, but directly traced to source code. Additionally, some trace patterns found in the dataset are more complex, than the ones discussed in [6], e. g. traces span more than four intermediate artifacts. That is why much *manual work* was required to identify appropriate entities and *match rules* in order to find them automatically in the dataset.

We noticed that graph structures in the studied projects evolve during project lifetime. Therefore, we analyzed metrics for a number of years for every project to track how metrics and their relation to bugs changed with the growth of the projects. The structure of graph became known through the history of the project until a certain date. Relations between objects are the vital thing that we need to process effectively to determine metrics of our interest. We aimed at tracing the relations between requirements to certain files through committed changes (and possibly, several other intermediate requirements) and from these files further to appropriate bugs through other committed changes. The approach is shown in Figure 2. Some of the files may be the same for a bug and a requirement. In this case we counted a bug against each requirement.

In order to simplify work with graph analysis, we investigated several databases, especially graph databases. Eventually, we chose Neo4J for our study. This decision was made based on the review of suitability of databases for software traceability discussed in [3]. Neo4J provides its own query language called Cypher. It is a declarative language similar in its syntax to SQL, while some rules and usages differ. There are two reasons for that: Cypher is a young language in comparison to SQL and it operates on data with different characteristics. The operations are performed on nodes and edges of graphs, instead of tables, tuples and relations with keys or foreign keys.

3 Data Transformation

The used "IlmSeven Dataset" [5] uses relational databases to store the artifacts and traces. Thus, the contained data needed to be transformed and mapped to the basic building blocks of Neo4J: nodes and relations. Entities from the relational schema became entities of the graph schema and foreign keys of the relational schema were transformed into relations in the new schema. Although this was an entirely technical task, considerations on effective logical and physical database schemas for Neo4J queries were required. Attribute values of the relational tuple were encoded as node properties in the new schema.

At first, we defined a mapping from ITS issue types, to the conceptual types "epic", "feature", "story" and "task" used for trace metric calculation. This information was used in import procedure to assign proper node types to requirements in the new schema. Bugs were represented as nodes with label "Bug". Unrelated issues such as test cases were filtered during import. In [4] the authors showed, that the artifact interlinking in open-source projects is not complete. We therefore excluded all artifacts that did not have relations to other issues or commits, because our trace metrics required connected artifacts. For issues different from type Bug, relations of type `RELATED_TO` are used to model the trace, possibly spanning multiple intermediate non-bug nodes. Only nodes representing change nodes are directly connected to file nodes in the graph. Nodes of type bug connect to other issue nodes with the edge type `RELATED_BUG`. All issues, including nodes of type bug, are connected to change nodes using the edge type `RELATES`.

A crucial part of data preparation was filtering only appropriate issues and bugs that are linked to changed files and make meaningful changes. This was based on manual study of textual information in sample tuples in the database. Beside problems in mapping the data from a relational to a graph structure, we also faced minor technical difficulties during data import into Neo4J database and they were successfully overcome.

4 Lessons learned

We gained a lot of experience from successful data mapping, transformation and metric calculation which will be discussed in the next paragraphs. At first, we show different queries written in Cypher query language illustrating the advantages of using a graph database for traceability analysis tasks.

Lesson 1: The beauty of concise path queries. To illustrate the flexibility and ease of Cypher query language, we consider the task of obtaining all incoming and outgoing links for every issue node in the graph:

```
MATCH (i:Issue)−[r1:RELATED_TO∗1..1]−(j:Issue)
WITH i, count(r1) as r2 SET i.connected=r2
RETURN count(r2);
```

As the example shows, complex structures are expressed easily with human-friendly SQL-like syntax. This reduces efforts for debugging, evolution and elaboration of queries.

Lesson 2: Cypher allows to compute and aggregate multiple variables in one query. All three considered trace metrics NRR, ADRR and RIF can be computed in one single query:

```
MATCH p1=shortestpath((a1:Issue)−
[:RELATED_TO∗1..]−(b1:Issue))
WHERE a1$<>$b1
WITH a1 AS Node, count(distinct b1)
AS NRR, sum(length(p1)) AS paths L,
sum(b1.connected) AS Flow
RETURN Node.issue id AS Issue, NRR,
toFloat(paths_L)/NRR AS ADRR,
toFloat(Flow)/NRR AS RIF ORDER BY Issue;
```

The query shows, that complex graph analysis tasks can be quite easy if proper tools are employed. Consequently, such task as preliminary study of relationships between traceability artifacts in data becomes much an easy task using Cypher.

Lesson 3: Complex tasks like traversing a graph can be performed transparently. We were interested in the number of bugs for a given requirement in our sample problem (see section 2). For instance, we considered traces from issue nodes to changes. This task can be achieved with the following Cypher query:

```
MATCH (i1:Issue)−[:RELATED_TO∗1..]−>
(c1:Change)−[fc1]−>(cc:File)<−[fc2]−(c2:Change)<−
[:RELATES∗1..]−(i2:Issue{type:"Bug"})
WHERE (i1.type<>"Bug") AND
(i2.created date>c1.committed date AND
(i2.created date−c1.committed date <31536000))
WITH i1,count(cc) as ncc, i2
RETURN DISTINCT (i1.issue id) AS requirement,
count(distinct i2.issue id) AS Bugs
ORDER BY requirement
```

The query considers two cases of relations between "issue" and "change" artifacts that are connected to each other through relations and a sequence of other issues or directly. In case issues are directly connected to changes, we only need to change the name of the relation and remove the nesting depth of relations in the beginning of the query:

```
MATCH (i1:Issue)−[:RELATES]−>(c1:Change)−[fc1]−>
(cc:File)<−[fc2]−(c2:Change)<−
[:RELATES∗1..]−(i2:Issue{type:"Bug"})
```

As can be seen, queries are easy to change.

Lesson 4: New technology also has problems. We observed, that Neo4J performance and effectiveness (i. e. the ability to finish a query) is better for smaller traceability graphs with simple structure and low interconnectivity, i. e. a small number of connections for a certain node. Performance problems of Neo4J graph database are also considered in [2]. Experiments with our dataset revealed, that complex queries, especially with sub-queries, grouping, correlated or nested queries are not always tractable by the system.

Furthermore, we experienced, that Cypher language of Neo4J in its current version 3.2, lacks flexibility and comprehensive description of its functionality. A lot of useful functionality was implemented in a third-party project *APOC library*[1] of user defined functions which.

We also encountered a number of obvious bugs in the Cypher language like the fact that preprocessor allows undocumented usage of patterns of relationships. For example, the name `RELATES` is expected to be a name of a variable in the query ()−>[RELATES∗1..5]−>(), but it seems to behave as a type of a relation. The documented pattern ()−>[:RELATES∗1..5]−>() means that `RELATES` is essentially a type of a relation. The only difference is the colon in front of the name that is easy to overlook. Development environment produces no warning message to assist the user.

At last, the *web frontend* of Neo4J has *too simple interface* and though is a perfect playground for testing queries, its usability leaves much to be desired, e. g. its editing facilities are not perfect, messages (or their absence) about syntax errors or query execution problems are confusing.

5 Conclusion

In this paper, we present our experience of using a graph database Neo4J for traceability analysis. Therefore, we selected a dataset stored in a relational database, and imported that data into Neo4J. Afterwards, we calculated three different requirements metrics, which are mainly based on trace pattern retrieval: a quite complicated task to perform with a relational database. Although we admitted deficiencies in Neo4J tools, we suggest usage of graph (hierarchical and network) databases for traceability problems. Based on our case study and implementation, we conclude that advantages of graph databases outweigh its deficiencies. Graph databases provide easiness and flexibility of querying data and query maintenance.

Acknowledgment

We are funded by the German Ministry of Education and Research (BMBF) grants: 01IS14026A,

[1] `https://github.com/neo4j-contrib/neo4j-apoc-procedures`

01IS16003B, by DFG grant: MA 5030/3-1, and by the EU EFRE/Thüringer Aufbaubank (TAB) grant: 2015FE9033.

References

[1] P. Mäder and J. Cleland-Huang. A visual traceability modeling language. In *Model Driven Engineering Languages and Systems - 13th International Conference, MODELS 2010, Oslo, Norway, October 3-8, 2010, Proceedings, Part I*, pages 226–240, 2010.

[2] A. Pacaci, A. Zhou, J. Lin, and M. T. Özsu. Do we need specialized graph databases? benchmarking real-time social networking applications. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, number 12 in GRADES'17. ACM, 2017.

[3] M. Rath, D. Akehurst, C. Borowski, and P. Mäder. Are graph query languages applicable for requirements traceability analysis? In *22nd International Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2017)*. IEEE Computer Society, 2017.

[4] M. Rath, M. Goman, and P. Mäder. State of the Art of Traceability in Open-Source Projects. *Softwaretechnik-Trends*, 37(3), 2017.

[5] M. Rath, P. Rempel, and P. Mäder. The IlmSeven Dataset. In *Requirements Engineering Conference (RE), 2017 IEEE 25th International*. IEEE, 2017.

[6] P. Rempel and P. Mäder. Estimating the implementation risk of requirements in agile software development projects with traceability metrics. In S. K. Fricker SA, editor, *Requirements Engineering: Foundation for Software Quality*, volume 9013. Springer International Publishing, 2015.

[7] P. Rempel and P. Mäder. Preventing defects: The impact of requirements traceability completeness on software quality. In *EEE Transactions on Software Engineering*. IEEE, 2016.