Supporting Program Comprehension through Fast Query response in Large-Scale Systems

Jinfeng Lin, Yalin Liu, Jane Cleland-Huang
University of Notre Dame
Notre Dame, IN
jlin6@nd.edu,yliu26@nd.edu,JaneHuang@nd.edu

ABSTRACT

Software traceability provides support for various engineering activities including Program Comprehension; however, it can be challenging and arduous to complete in large industrial projects. Researchers have proposed automated traceability techniques to create, maintain and leverage trace links. Computationally intensive techniques, such as repository mining and deep learning, have showed the capability to deliver accurate trace links. The objective of achieving trusted, automated tracing techniques at industrial scale has not yet been successfully accomplished due to practical performance challenges. This paper evaluates high-performance solutions for deploying effective, computationally expensive traceability algorithms in large scale industrial projects and leverages generated trace links to answer Program Comprehension Queries. We comparatively evaluate four different platforms for supporting industrial-scale tracing solutions, capable of tackling software projects with millions of artifacts. We demonstrate that tracing solutions built using big data frameworks scale well for large projects and that our Spark implementation outperforms relational database, graph database (GraphDB), and plain Java implementations. These findings contradict earlier results which suggested that GraphDB solutions should be adopted for large-scale tracing problems.

KEYWORDS

Software project queries, scalability, performance, traceability

ACM Reference Format:
Jinfeng Lin, Yalin Liu, Jane Cleland-Huang. 2020. Supporting Program Comprehension through Fast Query response in Large-Scale Systems. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3387904.3389260

1 INTRODUCTION

Software and Systems engineering projects accumulate large amounts of project data such as bug reports, feature requests, commit messages, design documents, code, test cases, and release documents. Project-wide queries issued against this data can place actionable intelligence into the hands of project stakeholders to support decision making, process improvement, compliance analysis, and numerous other software engineering tasks. Unfortunately, the heterogeneity of tool chains, and the lack of interconnecting trace links often makes it difficult for project stakeholders to retrieve and leverage project data in meaningful ways. In this paper we focus specifically on project queries that are designed to help software engineers understand the code, for example, understanding the state and functionality of a class, or building a conceptual model of the overall system architecture [15]. Several studies have identified common questions that developers ask [25, 39]. Questions such as "What does the declaration or definition of this look like?" can best be answered using features built into an integrated development environment (IDE) to explore the code base, while other questions such as "Which type represents this domain concept?", "How is this feature implemented?" or "Who has worked on this code?" often require queries that cut across artifacts such as feature descriptions (e.g., informal lists or more formal requirements specifications), commits, architectural documents, and source code files.

Most meaningful queries cut across multiple artifacts and therefore require underlying traceability links to be established. However, in many projects these links are either non-existent, inaccurate, or only partially available. Traceability is often perceived as desirable but prohibitively expensive, and is therefore often only adopted in safety-critical industries when it is required for certification purposes [14], and it is typically perceived as impractical to manually create and maintain a complete and accurate set of links in most industrial projects. To address this problem, researchers have proposed a slew of automated solutions based on information retrieval (IR) methods [26] such as topic modeling, Vector Space Model (VSM), Latent Dirichlet Analysis (LDA), classical machine learning, and deep-learning techniques. An increasing body of evidence suggests that combinations of tracing algorithms such as repository mining [33], deep learning techniques [22], and other computationally intensive techniques [17] provide the greatest promise for delivering accurate trace links.

1.1 Problem Statement

Despite the importance of requirements traceability for servicing diverse queries, researchers have focused their efforts on delivering novel algorithmic solutions while almost entirely ignoring the technical challenges related to high *scalability* and *performance* that are essential for deployment in large industrial projects. In fact, our work in this paper is motivated by two recent engagements with multi-national organizations. The first organization (ORG-1) specializes in the manufacture of telecommunications equipment and consumer electronics; while the second (ORG-2) is a US based

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389260

company specializing in the aerospace industry. Both organizations have emphasized the need for achieving scalable traceability to support a diverse set of project-wide queries. In both cases, they need to create, maintain, and use millions of trace links between diverse types of artifacts – several of which are constantly changing and evolving. The real-life scenario of traceability in industry to support high-performance queries is therefore a far cry from the majority of research projects which have focused on much smaller and relatively static ecosystems of software artifacts.

1.2 Contribution

In this paper we address two key aspects of querying software projects from the scalability perspective, namely *automating trace link generation* to provide just-in-time support for queries, and issuing *project-level queries* over a large set of existing set of trace links. Our goal is to achieve sufficiently fast trace link generation and query response times in large-scale industrial projects, so that ad-hoc project queries can be seamlessly integrated into the typical development workflow. According to Miller et al. [28], users notice a delay if the response time for a query is greater than one second; however, they are able to maintain focus on their task as long as the response time is less than 10 seconds. Response times that exceed 10 seconds require a progress bar and other mechanisms to keep the user engaged. While, trace queries can be generated overnight using batch processes, in practice, project stakeholders often wish to execute trace queries as part of their regular workflow. Therefore, our goal is to achieve response times within 10 seconds or less on large, industrially sized projects. In this study, we show that by using high-performance platforms the traceability algorithms and query solutions, which have been produced by the research community over the past decade, can satisfy practical performance needs in large industrial projects. Based on the experimental results in the first parts of this paper, we then investigate performance of traceability in a large-scale electronic health-care system.

The remainder of the paper is laid out as follows. Section 2 describes related work in high-performance traceability and briefly summarizes work in the area of querying software projects. Section 3 then lays foundations for the remainder of the paper by providing an overview of environments and frameworks that are used in our experiments. Sections 4 and 5 describe experiments for comparing the performance of various frameworks and platforms for trace link generation at scale and issuing project queries across existing trace links. In Section 6 we explore and discuss the application of our high-performance tracing and querying solutions in an open-source large-scale industrial Electronic Health-Care system consisting of over 36,878 code files and 1173 requirements. Finally, in Sections 7 and 8 we discuss threats to validity and conclusions.

2 RELATED WORK

Very few researchers have addressed performance issues related to traceability. However, there are some exceptions. Elamin et al., [18] explored the use of a graph database (GraphDB) as an alternative to a relational database for building a traceability repository. But their experiments were conducted on three relatively small datasets, as the largest one only including 371 nodes and 716 links. Their results showed that GraphDB outperformed a relational database, but still provided no evidence that GraphDBs were capable of handling trace queries at the millions-of-artifacts scale requested by our industry collaborators. Furthermore, they did not investigate the use of big data frameworks or plain Java as external query execution engine.

Other researchers have developed specific traceability tools and analyzed the performance of these tools. Sherba [37] developed 'InfiniTe' to support automated traceability within an industrial setting. The tool represented artifacts, such as requirements and code, in an XML format and then used XLink to represent trace links. Queries and link evolution algorithms were built over the XML layer using Java. Scalability was evaluated on a dataset containing 1,236 Perl source code files and 4,603 e-mail messages containing development discussions. It took almost five minutes to import source code and over 26 minutes to import emails, and an additional 3.5 hours to generate links. While this was performed on a machine in 2005, it is clear that the overall solution does not meet our performance goal for 10-second response time. More recently Sinha et al. [40] used a multi-graph structure to represent artifacts and relationships among them and then generated trace links over this structure. They reported a fast response time of 45 seconds to generate 7 million trace links; however, their approach is based on a premise that artifacts are constructed with embedded name tags. This form of traceability is costly to create and maintain, and this severely constrains its utility for most enterprise environments.

Several researchers have previously investigated the kinds of questions practitioners would like to ask. Fritz et al. [21] interviewed professional developers and identified questions related to source code, change sets, teams and work items. They categorized the questions into groups such as Who is working on what?, Changes to the code , and Work item progress. Other researchers conducted similar studies and identified queries related to software development, code change tasks, traceability usage scenarios, and data science [10, 24, 27].

3 AVAILABLE PLATFORMS

Our work is inherently constrained by the availability of platforms and environments in which to experiment. Therefore, before describing our experiments, we provide a brief overview of the platforms and the frameworks that we used in our evaluation.

3.1 Relational vs. Graph Database

Relational databases have been widely used in industry to persist data. They support SQL as a native relational query language. In the traceability domain, TIMs model the underlying artifact schema of a project, and tend to satisfy the Third Normal Form (3NF). In a relational database, artifact types are represented by tables, while individual artifacts are represented as records in the table. Trace links form junction tables containing at least two columns, where each column refers to the primary key in an artifact's table and forms a foreign key constraint. Trace queries are executed over this schema by leveraging SQL operands such as join, filtering and aggregation and where necessary combining intermediate results from sub-queries. We selected MySQL as the representative relational database platform because it is one of the most popular DBMSs (database management systems) with a free-license [2], offers competitive features, and often outperforms non-free DBMSs

[5]). Although some studies have shown that SQL Server can be up to 80 times more efficient than MySQL on select-join tasks [1], our analysis, reported later in this paper, indicates that our decision to use MySQL does not impact the overall ranking of our results.

GraphDBs store artifacts as vertices with associated attributes. Trace links are then represented as edges with optional attributes that can be used to store information, such as link similarity scores. GraphDB queries are written in a dedicated graph query language (GQL). We selected Neo4j to represent GraphDBs as it is one of the most popular industry options for large-scale datasets. It was also the platform specifically requested by our collaborators (ORG-1).

3.2 Big Data Processing Frameworks

Big data processing has been characterized by the three V's of Volume, Velocity, and Variety [35]. Industrial traceability fits all three of these characteristics because it addresses millions of artifacts and multi-millions of links (volume), artifact types, such as code and execution logs, that can grow and evolve rapidly (velocity), and software data that is typically stored in heterogeneous formats within diverse and case tools and repositories (variety).

Traceability researchers have typically processed entire sets of trace links as batch processes; however, in practice, these links need to be processed upon demand. Big data frameworks support several approaches to processing including batch, near-time, real-time, and streams. Well-known frameworks such as Apache Hadoop [42], HTCondor [41], and Pegasus [4] use batch processing to optimize for high throughput at the cost of higher latency; while frameworks such as Apache Spark [7] perform real-time and near real-time computations. Finally, Apache Flink [20] and Apache Storm [19] focus on stream processing to provide event-driven computation for emerging IoT applications. However, given the need for trace queries to execute quickly without interfering with the user's workflow, we opted to use Apache Spark for our experiments.

Spark provides several additional benefits. First, it includes a rich set of libraries such as Spark SQL [6] and SparkML [3] which can significantly reduce developers' effort for customizing trace models and generating queries. While stream processing systems are lightweight and provide real-time support, traceability data is not so volatile as to require this level of support. Second, Spark datasets leverage a table-like data structure, making them compatible with common SQL operations, such as JOINs and SELECTs. Spark tables are distributed into horizontal partitions, and parallelism is achieved by mirroring operations across those data partitions. Furthermore, Spark workers can be either an individual machine or a processor, meaning that Spark applications can run on a single multi-processor workstation or on a massive cluster.

4 TRACE LINK GENERATION AT SCALE

Automated trace link generation is intended to alleviate the high cost of manually creating and maintaining a sufficiently complete and accurate set of links to support meaningful project queries. Recent work has shown that generating trace link using a combination of information retrieval and repository mining techniques [33] can return highly accurate results at little effort to the user. However, the vast body of traceability research has focused on improving link quality while ignoring the performance challenge of scaling up

Table 1: A Sample of Traceability Datasets provided by CoEST.org

Project	Description	Source Artifact	Target Artifact	Trace Links
Albergate	Hotel management system	17	55	54
CM1	NASA system	22	53	45
eAnci	Municipalities management	30	20	567
Easy Clinic*	UI for Clinic system	46	73	195
EBT*	Event Based Traceability	40	25	98
eTour	Tour guide system	58	116	308
MIP	Medical infusion Pump	127	21	131
SMOS	Student monitoring system	67	100	1044
WARC*	library to parse WARC files	21	89	89

* Project includes multiple artifact types, and those with most trace links are shown.

to larger data sets. Traceability experiments have been conducted on relatively small (or even trivially sized) projects as depicted in Table 1. Zogaan et al. [44] collected information about 73 datasets used in prior research publications. Of these, 31 were derived from open source software (OSS) projects, 24 from academic projects (e.g. student projects), and 18 from industrial projects. The largest of these datasets was taken from an industrial project with 4,845 issue reports, 6,104 non-code artifacts, 5,135 documented trace links, with 29,573,880 potential links. This dataset represents a meaningful industrial problem; however, the research based on the dataset focused on the use of trace links and not on performance [12].

We investigate the performance of generating trace links for large industrial datasets using big data processing frameworks. In this section, we report only on performance issues using commonly adopted tracing algorithms, for which accuracy metrics have already been extensively reported (e.g., [26]) and treat the performance issue as a crucial research topic in its own right, which must be addressed as a precursor to achieving greater trace accuracy in large datasets. For example, a prior work using Recurrent Neural Networks (RNNs) [22] showed significant improvements in tracing accuracy; however, those experiments reportedly took several weeks to execute. Tackling performance issues therefore represents a critical research step that could lead to breakthroughs in trace accuracy, and will ultimately open the way for advancing promising, yet computationally expensive, tracing algorithms for use in real-world projects. Later, in Section 6, where we apply the high-performance algorithms against an industrial dataset, we report both performance and accuracy results.

To evaluate the performance of trace link creation algorithms we compare a typical Java solution for generating trace links with a solution based on a big data framework, and structure this part of our work around the following research questions:

- **RQ1:** How quickly can a plain Java implementation generate trace links for a large dataset?
- **RQ2:** To what extent can big data processing platforms improve the performance of trace link generation compared to a plain Java solution?

4.1 Link Generation with Voting Ensemble

To explore the run-time performance of on-demand, automated trace link generation in real-world projects, we adopted a *Voting Ensemble* approach and applied it to the task of generating links between issues and commits. Voting ensembles combine the results from multiple tracing algorithms then use a voting technique to establish the final set of candidate trace links. They have been shown to return more accurate results than single techniques [17]; however, they are computationally expensive because each tracing technique must be executed individually prior to voting on each link. Our implementation of the voting ensemble includes the following commonly used preprocessors and tracing algorithms – each of which had a publicly available implementation.

- **Standard Preprocessing:** Following common information retrieval practices, all artifacts were preprocessed to remove stop words, stem words to their morphological roots, and split both camel-case and snake-case words (e. g., optionsParser vs. options_parser) into their constituent parts.
- **Vector Space Model (VSM):** VSM [36] has consistently performed well for automated trace link creation [8]. VSM represents each *query* q (i. e. the source artifact) as a weighted vector of terms in a multidimensional space $q = (w_{1,q}, w_{2,q}, \ldots. w_{t,q})$. Documents (i. e. target artifacts) are similarly represented $d_j = (w_{1,j}, w_{2,j}, \ldots. w_{t,j})$. Term t in document d is typically weighted using the *term frequency, inverse document frequency* (tf-idf) and computed as $w_{t,d} = tf_{t,d} \cdot log\frac{|D|}{|\{d' \in D | t \in d'\}|}$ where $tf_{t,d}$ represents the term frequency of term t in document d, $|D|$ is the total number of documents in the set, and $|\{d' \in D | t \in d'\}|$ represents the number of documents that contain term t. The similarity between query q and document d_j is estimated as the cosine of the angle between the two vectors.
- **N-gram:** N-gram, i.e., a contiguous sequence of n words, were used to enhance VSM [43]. Each document was represented as a vector comprised of both words and N-grams. The size of N-gram is typically set to 2-gram, 3-gram or 4-gram sequences in the vector representations. Similarity between vectors is computed in the same way as the VSM approach.
- **Latent Dirichlet Allocation (LDA):** LDA [11] represents a class of algorithms based on topic distribution and has performed well in practice. Each topic t is defined as a probability distribution θ_t (over W words) drawn from a Dirichlet distribution with parameter β. Each document d is then associated with a probability distribution θ_d, drawn from a Dirichlet with parameter α. Any document d passed into the trained LDA model produces a topic probability distribution vector v_d, where $v_d = ((t_1, \theta_{t_{1_d}}), (t_2, \theta_{t_{2_d}}), \ldots, (t_n, \theta_{t_{n_d}}))$ for n topics, where $\theta_{t_{i_d}}$ represents the likelihood the document contains the ith topic. The cosine similarity between two documents d_i and d_j is computed using the topic distribution vectors v_{d_i} and v_{d_j}.

Our voting ensemble incorporated preprocessors (i.e., stemmer, stopper, and splitter), tracing techniques (VSM, N-gram, LDA), and a voting mechanism. Each individual technique votes for its top ranked 5% of links, and as a result, each link receives a voting score ranging from 0 (no votes) to 3 (full support by all techniques). The ensemble was used to generate trace links between commit messages and issues, primarily representing feature requests and bug fixes.

Table 2: Environments for Trace Link Generation

Env.	Platforms	Details
Description: Apply a voting ensemble constructed from VSM, N-gram VSM and LSI components to generate trace links between issue and commit artifacts for the datasets in Table. 3. Execution time including artifact retrieving and in-memory link generation are compared for environments E1 and E2		
E1	MySQL 8 Java 8	The voting ensemble was implemented using plain Java and a Java library of tracing algorithms. Artifacts were retrieved from MySQL database and converted into Java objects.
E2	MySQL 8 Spark 2.1	Leveraged the Spark framework and its built-in libraries (e.g. SparkML which provides implementations for machine learning models), to construct the same voting ensemble as E1. Artifacts were retrieved from MySQL and stored in memory using a table-like Spark data structure.

4.2 Implementing Tracing Algorithms

Traceability researchers have typically focused on tracing from requirements to design or code, and have implemented tracing algorithms using diverse programming languages, such as Java, C++, Python, and C#. Many traceability research tools such as RETRO, Poirot, and TraceLab have all used these approaches. To create a baseline representing a typical research implementation, we therefore used a standard library of tracing algorithms, implemented in Java without extra multi-threading enhancements. We also incorporated two third-party libraries. OpenNLP [9] provided support for text processing tasks such as tokenization, N-gram and cleaning, while Gensim [34] provided features to support LDA and tf-idf, used by the VSM and N-gram VSM models.

4.3 Experimental Evaluation

Based on the available platforms and environments, we evaluated the link creation task using plain Java and a Spark (big data platform) environment. These are summarized in Table 2. We conducted all experiments for our two tracing scenarios on the same machine with the following hardware specifications: 1) Intel i7-6700K CPU with 4 cores, 2) 28G of DDR3 RAM, and 3) a 500G SSD. Each experiment was executed three times and the average results are reported in all performance-related figures and tables throughout the remainder of this paper.

We used software artifacts mined from seven open source projects as depicted in Table 3. Projects in this dataset contains three types of artifacts including the commits, issue summaries, and related discussions which were all retrieved using the Github Rest API, and the trace links of the projects were extracted using internal git IDs. We executed the voting ensemble workflow using plain Java (see E1, Table 2) and Spark (See E2, Table 2). Each environment was configured to ensure that it had sufficient memory to avoid disk I/O during the execution, and native optimization techniques were

Table 3: Datasets used for Experiment 1

Project	Description	Issue	Commit	Candidate Links
EasyReact[3]	Reactive prog.	67	99	6,633
San[2]	JavaScript comp.	37	1,029	38,073
Arouter[1]	Application builder.	494	192	94,848
Arthas[1]	Java diagnostics	331	400	132,400
Canal[1]	Database log	1,011	578	584,358
Rax[1]	Application builder	519	2,484	1,289,196
Bk-cmdb[4]	Config. Manage.	794	3,818	2,993,312

Organizations: Alibaba[1], Baidu[2], MeiTuan[3], Tecent[4]

Table 4: Performance using the voting ensemble model to generate trace links using plain Java versus the Spark platform. (Time in seconds)

	All Source Artifacts			Single Artifact
Dataset	Spark	Java	Speedup	Spark
EasyReact	83	18	0.224	1.76
San	75	108	1.440	1.61
ARouter	71	151	2.130	2.76
Arthas	70	221	3.160	1.93
Canal	79	1,252	15.810	5.88
Rax	92	3,948	42.610	1.92
Bk-cmdb	244	18,161	74.380	2.03

applied to obtain optimal performance. For the Spark environment, we allocated 1GB memory to the Spark driver process and 1GB of memory to each of the four worker processors, resulting in a total of 5GB of memory. This was sufficient given that Spark uses compact formatting for its data. For the Java program, we allocated all 28GB of available memory to the JVM to avoid outOfMemory exception.

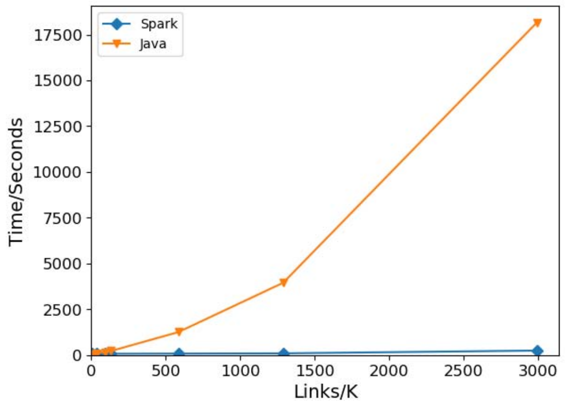


Figure 1: Performance of executing voting ensemble model on Spark and Plain Old Java

Results are reported in the first three data columns of Table 4, and show that in all but the smallest project (EasyReact), there was a significant speedup when using Spark in comparison to the plain Java implementation. We calculated the Speedup of Spark over Java using the formula $Speedup = Time(Java)/Time(Spark)$. On the Rax dataset, with approximately 1.2 million potential links, the tracing task took one hour to execute in Java but only 92 seconds with Spark. Similarly, on the Bk-cmdb dataset with approximately 3 million candidate links, it took over five hours with Java and approximately 4 minutes with Spark. Rank the projects by total number of potential links, Figure. 1 shows that the Java implementation is highly sensitive to data size with execution times rapidly increasing as the project size increases. In contrast, the Spark implementation has a relatively stable performance over all the projects. For small projects (e.g., EasyReact, San, ARouter, and Arthas) the execution time takes from 70 to 85 seconds, whereas for larger projects, Spark executes the query in significantly less time than the Java implementation.

To answer RQ1 we observe that the plain Java implementation of the voting ensemble performed unacceptably slowly on large datasets. The slow response time means that this approach is not viable for use within a project stakeholders' daily workflow, and would be detrimental to researchers who may need to run multiple variants of the experiment.

We also answer RQ2 by observing that Spark returned an average speedup of 37.3 and showed high stability as the data size increased to industrial scale. The response time using Spark on our largest dataset was 4 minutes. While this is still far above our 10-second response goal, it would present a viable solution if supported by progress bars and other UI support mechanisms. However, in the forth column of Table 4, we also reports the times taken using Spark to generate trace links for a single commit to all issues, using a pre-trained voting ensemble. This would be needed to support an ad-hoc trace query if no links were present. In this case, all links were generated within 10 seconds. The slowest case was observed for the Canal dataset at 5.88 seconds due to the fact that Canal has the largest number of issues to trace against.

5 PROJECT LEVEL QUERIES

Project stakeholders create queries against software project data to support diverse activities such as requirements validation, safety analysis, and project status reporting. For example, a user might request *a list of mitigating requirements associated with a specific component* in order to check that hazards have been sufficiently mitigated. Servicing such queries requires underlying trace links to either be generated dynamically, or previously to have been created and persisted – typically in the form of a trace matrix (TM) stored in a relational database such as Postgres, MySQL, or SQL Server. Trace queries are then specified using SQL, and the database engine creates a query execution plan that leverages JOIN operations and TMs as junction tables to navigate across multiple artifacts. JOINs are computationally expensive on large tables. As industrial project queries often incorporate multiple (N) artifact types that must be joined using $N - 1$ TMs, the performance of JOINS is particularly important. For example, one of our industrial collaborator's (ORG-1) had a sample query that requested a list of all *design requirements* tested by a specific *test plan* with at least one *text execution result* for a given *project*, and on average each of their sample queries included at least four artifact types. We therefore address the following research questions related to trace queries:

- **RQ3:** Does a GraphDB support efficient trace query execution at a large scale, and how does it compare to a relational database?
- **RQ4:** Do external query execution engines, such as big data processing frameworks and plain Java, outperform a database query execution engine?

5.1 Multi-hop Query with existing links

The multi-hop query represents the scenario in which a trace query is issued across a linear path of artifacts. Given three artifacts A_1, A_2, A_3 with trace links $A_1 \rightarrow A_2$ and $A_2 \rightarrow A_3$, a multi-hop query traverses all three artifacts i.e., $A_1 \rightarrow A_2 \rightarrow A_3$. Our experiment executed a query across bug reports (bugs), commits.The trace links, $Link(bug \rightarrow commit)$ and $Link(commit \rightarrow code)$, were automatically generated prior to the experiment using VSM. We executed the query: **Return a list of source code files which are related to open bug reports**, ranked the results by likelihood, and presented the top scoring results to the user.

5.2 Experimental Evaluation

We executed the multi-hop experiment on Dataset-2, consisting of issues, commit messages, and Java source files for four different open source projects used in a previous study [33]. We compared four different environments consisting of a relational database environment, a GraphDB environment, as well as a Plain Java and a Spark environment. The characteristics of these datasets are reported in Table 6.

Table 5: Environments for Multi-Artifact Project Query

Description: Execute a multi-hop trace query on the datasets depicted in Table. 6 and generate links along the trace path *Code* $\rightarrow$ *Commit* $\rightarrow$ *Bug*. Execution time including artifact retrieval and in-memory query execution were compared for environments E3,E4,E5 and E6		
Env.	**Platforms**	**Details**
E3	MySQL 8.0	Formulated the trace query as a SQL query and executed it directly on a MySQL database. Data was stored in MySQL.
E4	Neo4j 3.3	Formulated the trace query as a Cypher query and executed it directly on a Neo4j database. Data was stored in Neo4j.
E5	MySQL 8.0 Java 8	Retrieved artifacts and links from MySQL database, converted them into Java objects and organized them as a directed acyclic graph. Applied BFS algorithm to search trace path and retrieved results that satisfied the query conditions.
E6	MySQL 8.0 Spark 2.1	Retrieved artifacts and links from MySQL database and stored them in a table-like Spark data structure. Formulated the trace query with operands provided by Spark and executed it upon a local Spark service containing 4 worker processes.

Table 6: Dataset-2: Project-Query #2 with Multiple Artifact Types

Project	Description	Bug	Commit	Code	Link[1]	Link[2]	Tot Links
Seam2	Java Frame.	1,541	100	7,000	154k	700k	854k
Pig	Data anal.	2,097	2,016	1,835	4,227k	3,700k	7,927k
Maven	Build sys.	1,477	5,481	2,975	8,095k	16,306k	24,401k
Drool	Bus. Rules	1,881	6,923	9,597	13,022k	66,440k	79,462k

Candidate Link Type: BugReport-to-commit[1], Commit-to-code[2]

Table 7: Result of running sample trace query on four platforms with different datasets measured in seconds

	Spark	Java	GraphDB	GraphDB*	MySQL	SQL*
Seam2	3.40	1.82	7.03	5.30	163.88	2.05
Pig	4.48	25.34	17.52	7.22	296.21	37.00
Maven	4.90	38.75	37.04	7.09	355,90	44.34
Drools	5.40	59.92	113.83	9.37	1374.73	171.73

GraphDB* refers to GraphDB with warm cache. SQL* reports a theoretical speedup achievable using SQL Server as reported by some benchmarks. MySQL was used as our primary experimentation platform based on a key benchmark claim [5].

The trace query was formulated using both SQL and Cypher. For the Java query (See E5, Table 2), the artifacts and links retrieved from MySQL using JDBC were transformed into Plain Old Java Objects (POJO), and then organized into a graph structure composed of a vertex set G and an Edge set E using Java HashMaps as containers for G and E. This design ensured O(1) complexity for retrieving linked target artifacts for a given source artifact. We applied a Breadth-First Search (BFS) algorithm to navigate through the graph and collect trace paths as query results. For the Spark implementation, we leveraged Spark's SQL-like operands and table-like data structure, and loaded SQL tables as Spark Datasets. We then used filter() and join() operands to perform computations similar to a SQL query.

Spark and Java environments were configured as in Experiment 1. For the MySQL database, we allocated 8G memory to ensure the whole dataset could be loaded into memory, and created indices and foreign keys to connect artifact and link tables in order to accelerate join and filter operations. For Neo4j, we allocated 20G memory to the JVM heap and 500MB to the cache so that the graphs could reside in memory. Indices were applied on the vertices id, link id and link score to accelerate the query execution.

5.3 Results and Analysis

Results are reported in Table. 7. As expected, Spark achieved the best performance and SQL the worst across all four platforms. More surprisingly, the Java implementation had comparable response time to GraphDB. On the largest dataset (Drools), the Java query execution time was 52.6% of the GraphDB query execution time. However, when the same (or similar) GraphDB query was executed multiple times, its execution time significantly was reduced due to caching. Given a cold start, the GraphDB needs to load data into

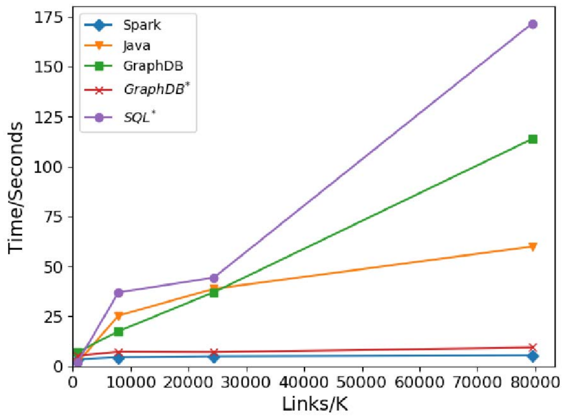


Figure 2: Trace query execution. GraphDB* refers to GraphDB query with warm cache. MySQL (without theoretical speedup) is not shown.

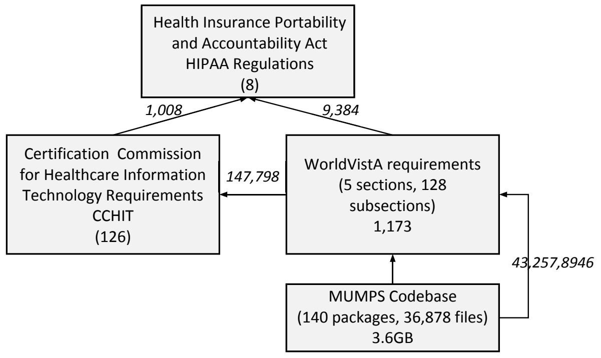
memory and create an internal graph which can then be reused for subsequent queries. To analyze the impact of this phenomenon, we depict the warm cached GraphDB as *GraphDB**, and include it in the comparison of all platforms in Figure 2.

Based on these results, we address our remaining research questions. For RQ3, we observe that the Neo4j environment was able to effectively execute trace queries on large datasets as long as sufficient memory was allocated. In practice, execution performance gradually increases as more trace queries are submitted to the GraphDB. As explained earlier, we selected the MySQL platform because benchmarks have reported that it is faster than SQL server for multi-join queries [5]. However, given other benchmarks that report SQL Server speedups for select-join queries we also evaluated the impact of applying the theoretically possible speedup of ×80 to the MySQL results. With these speedups, the results show SQL Server would have returned fast response time for smaller datasets, but was still significantly slower than both the cold-start and warm-start GraphDB as depicted in Table 7.

Finally, we address RQ4 which compares the use of external query execution engines (in our case Spark). The Spark performance was consistently faster than the fully-cached GraphDB across all four projects in the data set. Java provided the most efficient implementation on the smallest Seam2 project, but was less effective on the larger datasets compared to both Spark and *GraphDB**. We found that the majority of its execution time was spent creating POJOs and establishing a graph; while navigation over the established graph with BFS was very efficient and consumed less than 100ms for all projects. However, available memory limited the performance of the plain Java implementation, making it less than ideal as a platform for large industrial projects.

6 APPLICATION TO HEALTH-CARE SYSTEM

We evaluated our approach on the WorldVistA Electronic Healthcare Record (EHR) system, which was developed as an open-source system using the MUMPS programming language. MUMPS supports an integrated database environment that enables disk I/O using symbolic variables and subscripted arrays [23]. The WorldVistA project includes 1,173 textual requirements organized into five major sections and 128 subsections. These requirements were manually extracted from WorldVista documents by Rahimi et al.[32]. We visualize the project schema with a TIM model in Figure 3, the edges in this figure show counts of all potential links as no manually created links were available in this project for evaluation purpose. The WorldVistA codebase was downloaded from the Open Source Electronic Health Record Alliance (OSERA) github repository [31],



Health Insurance Portability and Accountability Act HIPAA Regulations (8)
1,008
9,384
Certification Commission for Healthcare Information Technology Requirements CCHIT (126)
147,798
WorldVistA requirements (5 sections, 128 subsections) 1,173
43,257,8946
MUMPS Codebase (140 packages, 36,878 files) 3.6GB

Figure 3: Traceability Information Model for the WorldVistA project showing artifacts used in our experiments. Italicized numbers show counts of all potential links.

HPS-PHA-111	Captures controlled substance dispensing to patients if electronic Controlled Substance Administration Record (CSAR) is implemented with Controlled Substance Version 3.0. (Health-Service Provider: Pharmacy Outpatient Services)
COS-MAM-008	Network Transmissions Over TCP/IP-Network transmissions can be made across Transmission Control Protocol / Internet Protocol (TCP/IP) channels to any Simple Mail Transfer Protocol (SMTP)-compatible mail system. (Common Services: Mailman))

Figure 4: Two WorldVistA requirements

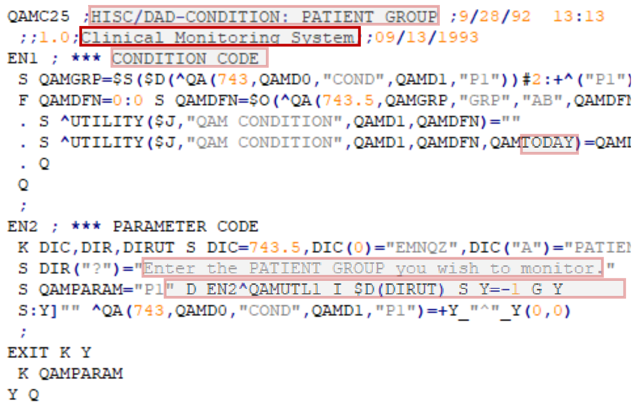

```
QAMC25 ;HISC/DAD-CONDITION: PATIENT GROUP ;9/28/92  13:13
 ;;1.0;Clinical Monitoring System ;09/13/1993
EN1 ; *** CONDITION CODE
 S QAMGRP=$S($D(^QA(743,QAMD0,"COND",QAMD1,"P1"))#2:+^("P1")
 F QAMDFN=0:0 S QAMDFN=$O(^QA(743.5,QAMGRP,"GRP","AB",QAMDFN
 . S ^UTILITY($J,"QAM CONDITION",QAMD1,QAMDFN)=""
 . S ^UTILITY($J,"QAM CONDITION",QAMD1,QAMDFN,QAMTODAY)=QAMI
 . Q
 Q
 ;
EN2 ; *** PARAMETER CODE
 K DIC,DIR,DIRUT S DIC=743.5,DIC(0)="EMNQZ",DIC("A")="PATIEN
 S DIR("?")="Enter the PATIENT GROUP you wish to monitor."
 S QAMPARAM="P1" D EN2^QAMUTL1 I $D(DIRUT) S Y=-1 G Y
 S:Y]"" ^QA(743,QAMD0,"COND",QAMD1,"P1")=+Y_"^"_Y(0,0)
 ;
EXIT K Y
 K QAMPARAM
Y Q
```

Figure 5: A section of a 'Mumps' source code file. The code contains comments and variable names, as outlined in red. These meaningful terms are used for link generation.

includes 3.6 GB of data composed of 36,878 code files organized into 140 packages. In addition, WorldVistA was certified by the Certification Commission for Healthcare Information Technology(CCHIT), which introduces an additional layer of 126 CCHIT requirements. Two sample requirements are shown in Figure 4 and a section of a Mumps source code file, with comments and variable names hightlighted, is depicted in Figure 5. Mumps makes extensive use of symbols and IDs which are not useful for linking to requirements; therefore we preprocessed the text by removing non-alphabetic characters, stopwords, and single character tokens (e.g., S,F, and Q in example code), and then tokenized the remaining text.

One of the major goals in this part of our study was to investigate whether the use of high-performance computing solutions could make interactive trace queries viable for large software projects. Given that the code-base is one of the largest parts of a project's dataset, and given the strategic importance of analysing and understanding source code, we focus this analysis on queries related to the source code.

6.1 System-wide traceability

As with our previous experiments, we explored the performance of link generation and trace query execution. We used VSM and LDA to generate trace links among Requirements (REQ), CCHIT Requirements (CHT), HIPAA Goals (HP), and Source Code (SC). Links sets included $SC \rightarrow REQ$, $REQ \rightarrow CHT$, $HP \rightarrow REQ$, and $HP \rightarrow CHT$. We implemented the same trace workflow with both (1) Plain-old-Java and (2) Spark and reused the trace workflow and experiment setup described in Sec.4.3, except that we allocated all 28G available memory to JVM for both POJ and Spark, because the source code in WorldVistA is significantly larger than the OSS used in our earlier experiments. To avoid memory exceptions and thrashing due to garbage collection, we partitioned the source code into five approximately equal parts, and traced to each part in turn. For LDA we used 50 topics and 100 training iterations.

Table 8: Time in seconds to generate trace links between WorldVista Requirement (REQ), CCHIT Requirements (CHT), HIPAA Goals (HP), and Mump Source Code files (SC)

Trace Task	Links	Plain Java		Spark	
		VSM	LDA	VSM	LDA
SC->REQ	43,257,894	2200.25	> 10 hr	105.10	1643.02
REQ->CHT	147,798	7.6	312.39	1.72	300.45
HP->REQ	9,384	0.2	31.38	1.22	107.99
HP->CHT	1,008	0.08	13.30	1.23	105.34

In this experiment, we included the time of loading artifacts from disk to memory. As reported in Table 8, link generation time increases as the number of candidate links increase. The results show that VSM implemented in plain Java performs best on smaller tasks (e.g., ($HP \rightarrow REQ$, $HP \rightarrow CHT$)) as it has less overhead; however, as the number of artifacts and potential links increases, Spark achieved speedups of 4.41 and 20.95 over plain Java. Using LDA with an iterative training step, was computationally expensive in comparison to VSM and Spark, and exhibited even more overhead than VSM. Consequently, Java achieved better performance over Spark on tasks $HP \rightarrow REQ$ and $HP \rightarrow CHT$; however, Java-based LDA was unable to finish the largest tracing task (i.e., $SC \rightarrow REQ$ task) after 10 hours of execution, which makes it impractical for use for tracing large artifact sets. In contrast, Spark LDA can complete this large task within 1,643.02 seconds (27.38 mins).

Table 9: Lists of keywords for retrieving architectural requirements and security related code files

Query Type	Keywords
Platform	system database client server distributed SQL SOA
Inconnectivity	messaging messages HL7 protocol synchronous asynchronous tcp transmit send transmission receive interface memory router exchange transport API
Security	authenticate authen credential kerberos login otp credential permission access security secure

We now explore how these performance results impact practical applications in the area of program comprehension in WorldVistA. We base our analysis on two queries, both of which leverage the links and tracing algorithms described in Section 6.1, in these two queries we evaluate the execution time of applying Spark based implementation where artifacts already reside in memory.

6.2 Query 1: Architectural Requirements

A significant body of work exists at the intersection of requirements engineering and architectural design. For example, researchers have explored the impact of requirements upon architecture [30] and design [13], including links between key stakeholder concerns, architecturally significant requirements, important architectural decisions and sections of code where architectural decisions are implemented [29]. Building on these concepts, our first WorldVistA query identifies the set of requirements that describe platform-specific, or inter-process communication features, which have cross-cutting impact across the system's source code. Query results provide insights into the organization of the code – for example, they differentiate between architectural requirements that are implemented in a modular versus a cross-cutting way, and identify specific packages which are impacted by multiple cross-cutting concerns. We performed a multi-step query that included the following steps:

(1) We created two search queries from keywords related to *platform* and *interconnectivity* respectively (see Table 9). We then used VSM to retrieve requirements relevant to each query. Any requirement with a similarity score above 0.05 was retained. We refer to these requirements as ASRs (Architecturally significant requirements).
(2) We then generated links from the ASRs to code using Spark-based VSM.
(3) For each requirement-package combination, we computed a *RequirementsPackageScore* by summing the similarity scores from requirements to source code files in the package.
(4) For each requirement, we then computed an *AggregatedPackageScore* by counting the number of packages for which it received a *RequirementsPackageScore* greater than a threshold value.
(5) We sorted the requirements in descending order by *AggregatedPackageScore.*
(6) Finally, we visualized the results as shown in Figure 6.

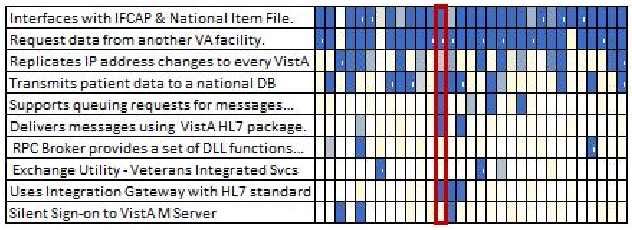


Figure 6: Cross-cutting architecturally significant requirements, retrieved by Query #1 are visualized on a color scale: blue (most impact), yellow (some impact), white (no impact). Shaded rows depict requirements with high cross-cutting impact. Shaded columns depict packages impacted by multiple architectural concerns. Here we show only 10 of 1,173 sample requirements and 31 of 140 packages.

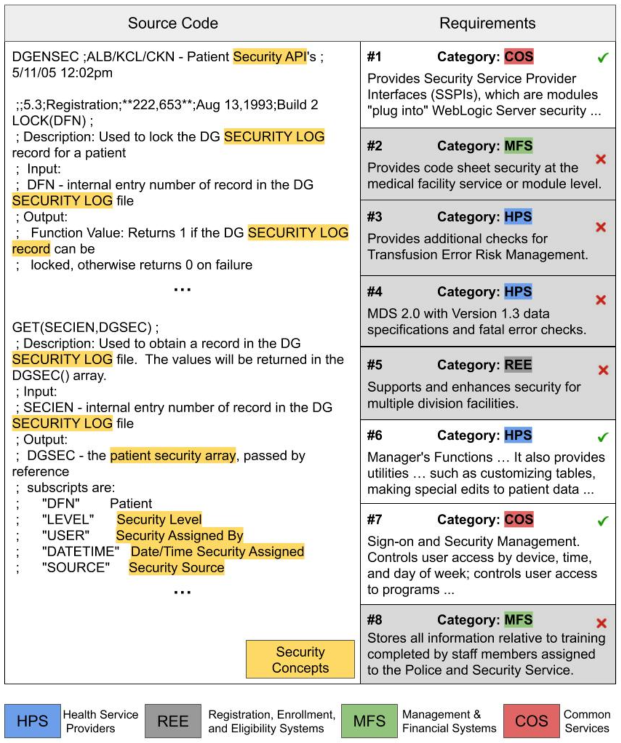
Results: We show results for ten crosscutting requirements in Figure 6, where requirements are depicted as rows, and packages as columns. The first two requirements (top two rows) are highly cross-cutting, while others exhibit varying degrees of cross-cutting characteristics. Our goal is to make trace queries sufficiently responsive so as to support interactive exploration of the code-base. Developers, or other users, could adjust various threshold values, for example, those associated with computing the *RequirementsPackageScore* to interactively explore the way requirements impact different source-code packages. They could then zoom into individual packages, vet the automatically generated links, and analyze which code files are impacted by the requirement.

Spark-VSM took 1.67 seconds to trace the two architectural queries to 1,173 requirements and then to select 48 potentially architecturally significant requirements. It then took 4.40 seconds to trace from these 48 candidate requirements to all source code files. Finally, it took 2.27 seconds to calculate the *RequirementsPackageScores* and the *Aggregated-PackageScores*. This resulted in a total execution time of 8.34 seconds, which is within the bounds of our previously discussed ten second query-response goal.

6.3 Query 2: Analyzing Security Code

When developers maintain and evolve existing source code, they need to carefully analyze it to understand what it currently does. They review source code, read comments, explore the code's interactions with other sections of code (e.g., its call hierarchy), and when available, review requirements associated with the code.

In this query we focus on security-related source-code files, and inspect requirements that are related to that code. Queries of this nature could be used for several reasons. First, a developer working in a code file, might wish to identify all of its relevant requirements, to gain a deeper understanding of the code's purpose. Second, as part of a strategic plan to modify the project-level access control, an architect or developer might want to identify and analyze all security-related code files, and also explore their cohesion with respect to security versus other functionality. As with Query 1, our goal is to support this type of query interactively, in an environment where the developer experiences sufficiently fast response time.



Source Code

```
DGENSEC ;ALB/KCL/CKN - Patient Security API's ;
5/11/05 12:02pm

 ;;5.3;Registration;**222,653**;Aug 13,1993;Build 2
LOCK(DFN) ;
 ; Description: Used to lock the DG SECURITY LOG
record for a patient
 ;  Input:
 ;  DFN - internal entry number of record in the DG
SECURITY LOG file
 ; Output:
 ;   Function Value: Returns 1 if the DG SECURITY LOG
record can be
 ;   locked, otherwise returns 0 on failure
                    ...

GET(SECIEN,DGSEC) ;
 ; Description: Used to obtain a record in the DG
SECURITY LOG file.  The values will be returned in the
DGSEC() array.
 ; Input:
 ; SECIEN - internal entry number of record in the DG
SECURITY LOG file
 ; Output:
 ;  DGSEC - the patient security array, passed by
reference
 ;  subscripts are:
 ;      "DFN"        Patient
 ;      "LEVEL"      Security Level
 ;      "USER"       Security Assigned By
 ;      "DATETIME"   Date/Time Security Assigned
 ;      "SOURCE"     Security Source
                    ...
```

Security Concepts

Requirements

#	Category	Requirement	
#1	COS	Provides Security Service Provider Interfaces (SSPIs), which are modules "plug into" WebLogic Server security ...	✓
#2	MFS	Provides code sheet security at the medical facility service or module level.	✗
#3	HPS	Provides additional checks for Transfusion Error Risk Management.	✗
#4	HPS	MDS 2.0 with Version 1.3 data specifications and fatal error checks.	✗
#5	REE	Supports and enhances security for multiple division facilities.	✗
#6	HPS	Manager's Functions ... It also provides utilities ... such as customizing tables, making special edits to patient data ...	✓
#7	COS	Sign-on and Security Management. Controls user access by device, time, and day of week; controls user access to programs ...	✓
#8	MFS	Stores all information relative to training completed by staff members assigned to the Police and Security Service.	✗

HPS	Health Service Providers
REE	Registration, Enrollment, and Eligibility Systems
MFS	Management & Financial Systems
COS	Common Services

Figure 7: One of the security files retrieved by Query #2 is shown on the left, while the eight requirements traced to this file are shown on the right. We accepted three of the requirements as valid and rejected the remaining ones.

In large projects, portions of source code are relatively stable while other parts are more volatile due to maintenance activities or the introduction of new features. While the reuse of existing trace links can significantly reduce query response time, it can also adversely impact accuracy – especially for the more volatile parts of the code. To achieve fast response time, an underlying trace model can be periodically updated in the background. In this query, we reuse our pre-built VSM tracing model which contains pre-trained term weightings; however, the trace links are generated upon demand. Our query involved the following steps:

(1) We created a query for retrieving security-related source code, using a subset of keywords collected by Mirakhorli et al. [29] for detecting security related sections of code. These keywords are shown in Table. 9.
(2) We reused the VSM model trained earlier to trace between $SC \longrightarrow REQ$ links in Sec. 6.1 in order to generate links from the security query to code.
(3) We classified security-related source code files, as files exhibiting a similarity score >= threshold T (0.05).
(4) We used the same pre-trained VSM model to generate trace links from these security-related code files to requirements. We accepted all requirements exhibiting a similarity score >= threshold P (0.05), and sorted the results in descending order of relevance.

(5) We visualized the results as depicted in Figure 7 This functionality is implemented in our automated tool (not shown here due to space constraints).

Results: Given the established thresholds, our query returned seven source code files for which we accepted three, associated with secure patient record access API, incoming message processing, and MD5 Hash Testing, were accepted as valid. We envision delivering interactive traceability tools with fast response times that would allow developers to interact with the results, raising and lowering thresholds to explore additional results upon demand. For each source code file, the developer would be able to browse and analyse its associated requirements as depicted in Figure 7. In this example, a source code file associated with secure access to patient data is displayed on the left hand side, while candidate requirements are shown on the right. We accepted three associated requirements as valid following a manual inspection. These requirements defined explicit security features (1,7), and utilities for editing patient data which would require security features (6). This query was designed to evaluate the modularity of security features, and an inspection of the retrieved security files, revealed high cohesion and relatively low coupling with non security-related features.

We divided the query into two sub-queries and measured the execution time for each of them. The results are shown in Table. 10. For the first sub-query (i.e., $SQ \rightarrow$SC) Spark took 6.5 seconds while Plain Java took 8.74 seconds to trace the security terms to 36,878 source code files. We then filtered the files according to a threshold similarity score, and this decision significantly impacted the execution time of the second query. For example, at one threshold setting, we retrieved 255 source code files (i.e., top 0.69% of files) and the subsequent query took 19.84 seconds in Java versus 8.77 seconds in Spark, which finished within our 10 second goal. With a much higher threshold that retrieved only 37 source code files (i.e., top 0.10% of files), Spark took 1.77 seconds while Plain Java took 1.87 seconds. As it makes sense to retrieve more files and allow the developer to interactively explore the relationship between source-code and requirements by adjusting the thresholds, the Spark solution, which scales up for larger artifact sets, seems to be the best option.

6.4 Implications of WorldVistA results

For traceability to be beneficial in industrial settings, it needs to return results sufficiently fast to allow users to interactively explore the results [16, 38]. Our study with WorldVistA showed the viability of achieving such response-times for an industrial system. Despite a large code-base of over 36,878 files, queries were executed within our targeted response time and returned results that could be interactively explored and refined [38].

Table 10: Time (secs) to trace a security query (SQ) to source code (SC) and associated requirements (REQ) using a prebuilt VSM model

Trace Task	Links	Plain Java	Spark
SQ->SC	36,878	8.74	6.5
SC->REQ	284,835	19.84	8.77
	41,329	1.87	1.77

7 THREATS TO VALIDITY

Performance evaluations for categories of solutions (e.g., GraphDBs vs. relational database) can suffer from the validity threat that the platform selected for the experiments might not be the best-performing representative for its category, or that the platform might not be configured in an optimal way. To mitigate this risk we carefully selected each of our platforms and configured them following many initial experiments. In cases where the selection might be considered controversial (e.g., due to conflicting benchmarks for MySQL versus SQL Server), or unduly influenced by operating context (e.g., GraphDB's cold-start problem), we have reported results for each case.

Second, our initial experiments were limited by available data. In industrial practice many trace queries include software and systems requirements; however, acquiring industrial data at the scale and diversity needed for multi-hop experiments is challenging and is likely the reason for the lack of performance research. Therefore, we conducted our initial experiments using artifacts such as feature requests and bug reports from OSS. We then partially mitigated this threat to validity by exploring the execution time of two non-trivial queries in WorldVista – a large OSS with an available set of requirements. Results from this analysis were aligned with those obtained from our initial experiments.

Finally, while we have proposed high-performance tracing environments that would support developers in their program comprehension tasks. It was out of the scope of this paper to conduct an actual user study. However, as our traceability and some of our interactive visualizations are already automated (e.g., Figure 7), we plan to conduct user studies once the environment is fully deployed.

8 CONCLUSION

The work reported in this paper is motivated by the industry challenge of generating and using trace links at scale with fast response-times, within large project environments. Our experiments showed that Spark, a big data framework, provided an average speedup of 37.2 over a plain Java implementation and was able to generate almost three million links in just over 4 minutes, and that it could generate links from a single issue (e.g., feature request or requirement) using a pre-trained voting ensemble in less than 6 seconds for each of our datasets. When executing queries over an existing network of trace links, both Spark and the warm-cache GraphDB achieved our 10-second response-time goal; however, Spark was consistently faster, and neither plain Java nor a relational database solution achieved our performance goals. In future work, we will explore the combination of a GraphDB with a Spark framework. However, our results suggest that organizations should not rely entirely on a GraphDB environment such as neo4J as additional fast processing capabilities are needed to generate new trace links on demand. Our datasets used for experiments are available on Zenodo[1]

ACKNOWLEDGMENTS

This work was partially funded under US National Science Foundation Grants CCF:1649448 and CNS:1649008.

[1]https://tinyurl.com/yxxkat9p

REFERENCES

[1] [n.d.]. Difference Between SQL Vs MySQL Vs SQL Server. https://www.softwaretestinghelp.com/sql-vs-mysql-vs-sql-server. Accessed: 2020-10-30.

[2] [n.d.]. Engines. https://db-engines.com/en/

[3] [n.d.]. Spark MLib Modlue. https://spark.apache.org/mllib.

[4] E. Deelman et al. 2005. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming* 13 (2005), 219–237.

[5] K. Islam et al. 2017. Huge and Real-Time Database Systems: A Comparative Study and Review for SQL Server 2016, Oracle 12c & MySQL 5.7 for Personal Computer. *Journal of Basic and Applied Sciences* 13 (2017), 481–490.

[6] M. Armbrust et al. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD)*. ACM, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

[7] M. Zaharia et al. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (2016), 56–65. https://doi.org/10.1145/2934664

[8] S. Lohar et al. 2013. Improving trace accuracy through data-driven configuration and composition of tracing features. In *European Software Eng. Conference (ESEC/FSE)*. 378–388. https://doi.org/10.1145/2491411.2491432

[9] Jason Baldridge. 2005. The opennlp project. *URL: http://opennlp. apache. org/index. html,(accessed 2 February 2012)* (2005), 1.

[10] Andrew Begel and Thomas Zimmermann. 2014. Analyze this! 145 questions for data scientists in software engineering. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 12–23. https://doi.org/10.1145/2568225.2568233

[11] David M Blei, Andrew Y Ng, and Michael I Jordan. 2003. Latent dirichlet allocation. *Journal of machine Learning research* 3, Jan (2003), 993–1022.

[12] Markus Borg, Orlena CZ Gotel, and Krzysztof Wnuk. 2013. Enabling traceability reuse for impact analyses: A feasibility study in a safety context. In *Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*. IEEE, 72–78.

[13] Lawrence Chung, Daniel Gross, and Eric S. K. Yu. 1999. Architectural Design to Meet Stakeholder Requirements. In *Software Architecture, TC2 First Working IFIP Conference on Software Architecture (WICSA1), 22-24 February 1999, San Antonio, Texas, USA*. 545–564.

[14] Jane Cleland-Huang, Orlena Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. 2014. Software traceability: trends and future directions. In *Proceedings of the on Future of Software Engineering, FOSE, Hyderabad, India, 2014*. 55–69. https://doi.org/10.1145/2593882.2593891

[15] Barthélémy Dagenais, Harold Ossher, Rachel K. E. Bellamy, Martin P. Robillard, and Jacqueline P. de Vries. 2010. Moving into a New Software Project Landscape. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. ACM, New York, NY, USA, 275–284. https://doi.org/10.1145/1806799.1806842

[16] Alex Dekhtyar and Jane Huffman Hayes. 2012. Studying the Role of Humans in the Traceability Loop. In *Software and Systems Traceability*. 241–261. https://doi.org/10.1007/978-1-4471-2239-5_11

[17] Alex Dekhtyar, Jane Huffman Hayes, Senthil Karthikeyan Sundaram, Elizabeth Ashlee Holbrook, and Olga Dekhtyar. 2007. Technique Integration for Requirements Assessment. In *International Requirements Engineering Conference, RE New Delhi, India*. 141–150. https://doi.org/10.1109/RE.2007.17

[18] R. Elamin and R. Osman. 2018. Implementing Traceability Repositories as Graph Databases for Software Quality Improvement. In *International Conference on Software Quality, Reliability and Security (QRS)*. 269–276. https://doi.org/10.1109/QRS.2018.00040

[19] R. Evans. 2015. Apache Storm, a Hands on Tutorial. In *IEEE International Conference on Cloud Engineering*. 2–2. https://doi.org/10.1109/IC2E.2015.67

[20] Ellen Friedman and Kostas Tzoumas. 2016. *Introduction to Apache Flink: Stream Processing for Real Time and Beyond* (1st ed.). O'Reilly Media, Inc.

[21] Thomas Fritz and Gail C. Murphy. 2010. Using information fragments to answer the questions developers ask. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. 175–184. https://doi.org/10.1145/1806799.1806828

[22] Jin Guo, Jinghui Cheng, and Jane Cleland-Huang. 2017. Semantically enhanced software traceability using deep learning techniques. In *International Conference on Software Engineering (ICSE)*. IEEE, 3–14.

[23] Norman F Hirst. 1976. MUMPS: Massachusetts General Hospital Utility Multi-programming System. *Medical Informatics* 1, 3 (1976), 163–165.

[24] Andrew J. Ko, Robert DeLine, and Gina Venolia. 2007. Information Needs in Collocated Software Development Teams. In *Proceedings of the 29th International Conference on Software Engineering (ICSE âĂŹ07)*. IEEE Computer Society, USA, 344âĂŞ353. https://doi.org/10.1109/ICSE.2007.45

[25] Thomas D. LaToza and Brad A. Myers. 2010. Developers Ask Reachability Questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. ACM, New York, NY, USA, 185–194. https://doi.org/10.1145/1806799.1806829

[26] Andrea De Lucia, Andrian Marcus, Rocco Oliveto, and Denys Poshyvanyk. 2012. Information Retrieval Methods for Automated Traceability Recovery. In *Software and Systems Traceability*. 71–98. https://doi.org/10.1007/978-1-4471-2239-5_4

[27] Sugandha Malviya, Michael Vierhauser, Jane Cleland-Huang, and Smita Ghaisas. 2017. What Questions do Requirements Engineers Ask?. In *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017*. 100–109. https://doi.org/10.1109/RE.2017.76

[28] Robert B Miller. 1968. Response time in man-computer conversational transactions.. In *AFIPS Fall Joint Computing Conference (1)*. 267–277.

[29] Mehdi Mirakhorli and Jane Cleland-Huang. 2016. Detecting, Tracing, and Monitoring Architectural Tactics in Code. *IEEE Trans. Software Eng.* 42, 3 (2016), 206–221. https://doi.org/10.1109/TSE.2015.2479217

[30] Bashar Nuseibeh. 2001. Weaving Together Requirements and Architectures. *IEEE Computer* 34, 3 (2001), 115–117. https://doi.org/10.1109/2.910904

[31] Osehra. 2019. OSEHRA/VistA-M. https://github.com/OSEHRA/VistA-M

[32] Mona Rahimi, Mehdi Mirakhorli, and Jane Cleland-Huang. 2014. Automated extraction and visualization of quality concerns from requirements specifications. In *IEEE 22nd International Requirements Engineering Conference, RE 2014, Karlskrona, Sweden, August 25-29, 2014*. 253–262. https://doi.org/10.1109/RE.2014.6912267

[33] Michael Rath, Jacob Rendall, Jin L. C. Guo, Jane Cleland-Huang, and Patrick Mäder. 2018. Traceability in the wild: automatically augmenting incomplete trace links. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 834–845. https://doi.org/10.1145/3180155.3180207

[34] Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. ELRA. http://is.muni.cz/publication/884893/en.

[35] Philip Russom et al. 2011. Big data analytics. *TDWI best practices report, fourth quarter* 19, 4 (2011), 1–34.

[36] G. Salton and M. McGill. 1986. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, NY, USA.

[37] Susanne A Sherba. 2005. *Towards automating traceability: an incremental and scalable approach*. Ph.D. Dissertation. University of Colorado at Boulder.

[38] Yonghee Shin and Jane Cleland-Huang. 2012. A comparative evaluation of two user feedback techniques for requirements trace retrieval. In *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*. 1069–1074. https://doi.org/10.1145/2245276.2231943

[39] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2006. Questions Programmers Ask During Software Evolution Tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Portland, Oregon, USA) *(SIGSOFT '06/FSE-14)*. ACM, New York, NY, USA, 23–34. https://doi.org/10.1145/1181775.1181779

[40] Roopak Sinha, Barry Dowdeswell, Gulnara Zhabelova, and Valeriy Vyatkin. 2019. TORUS: Scalable Requirements Traceability for Large-Scale Cyber-Physical Systems. *ACM Transactions on Cyber-Physical Systems* 3, 2 (2019), 15.

[41] Douglas Thain, Todd Tannenbaum, and Miron Livny. 2005. Distributed computing in practice: the Condor experience. *Concurrency and computation: practice and experience* 17, 2-4 (2005), 323–356.

[42] Tom White. 2009. *Hadoop: The Definitive Guide* (1st ed.). O'Reilly Media, Inc.

[43] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. 2016. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

[44] Waleed Zogaan, Palak Sharma, Mehdi Mirakhorli, and Venera Arnaoudova. 2017. Datasets from Fifteen Years of Automated Requirements Traceability Research: Current State, Characteristics, and Quality. In *Requirements Engineering Conf., RE*. 110–121. https://doi.org/10.1109/RE.2017.80