2021-08-27

Scalable Encoding of Modularized Dependency Graphs for Fast Analysis

Singh, Kanishka

UNIVERSITY OF CALGARY

Scalable Encoding of Modularized Dependency Graphs for Fast Analysis

by

Kanishka Singh

A THESIS

SUBMITTED TO THE FACULTY OF GRADUATE STUDIES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE

DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE

CALGARY, ALBERTA

AUGUST, 2021

Abstract

Software development and analysis tools (SDATs) typically contain complex models that are expensive to compute, and whose expense grows significantly depending on the size of the software system under analysis. When these models are not stored in a manner that allows them to be restored after program restart, that expense is not amortized; re-computation results in undesirable downtime in the developer's daily workflow. This thesis aims to find the most suitable approach for storing and persisting the models of a specific change propagation tool, ModCP. Existing work to study and identify optimal storage technology has been evaluated using datasets either that are randomly generated—not simulating the nature of real world software—or that derive from excessively small software systems for which recomputing would be feasible.

This thesis explores and implements potentially beneficial datastore technologies in ModCP and compares them on subjective and objective measures against the baseline (time to fully re-building the models) and each other to determine whether storage integration is feasible and significant reduction in the downtime can be achieved. The cost of rebuilding the model of ModCP can be reduced by 13–46 times, for the datasets we tried, by using specific serialization technology; in contrast, the use of database technologies involves high overhead for read/write queries through database connectors, making them unsuitable as an option for improvement of performance in SDATs.

In loving memory of my dog, whom I lost on this journey. You will always be in my heart.

To my father, for constantly nurturing and supporting me.

To my mother, for her unconditional love and believing in me.

Acknowledgments

I am deeply indebted to everyone who supported me during the completion of this thesis, especially Rahila Afzal who consistently encouraged me and was always there to support me in my hardest times. I cannot begin to express my thanks to my supervisor and guru, Dr. Robert Walker, for his constant guidance and support. The completion of my research would not have been possible without his help and direction.

I would like to express my deepest appreciation to my MSc defence examining committee members Dr. Reda Elhajj and Dr. Diwakar Krishnamurthy for kindly agreeing to be on my examination board. I am extremely grateful to Ken Gnazdowsky and Mike Cooper from Find it EZ Software for their assistance and valuable insights throughout my research. I would also like to extend my gratitude to Hao Men for his assistance during the beginning of my work.

Lastly, a special thanks to my brothers, Vansh and Kushagra, for their love and support throughout this journey.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 The problem

Real-world software systems are large; they are developed and evolved over time with changing business and technical environments by changing groups of people [Davis et al., 1988]. These changes can involve adding or removing functionality, stability improvements, security fixes, compatibility fixes, bug fixes, and improvements to the user experience, all of which are realized by modification, deletion, and addition to the existing software code base [Lehman et al., 1997]. While in principle a developer can make all needed changes with nothing more sophisticated than a text editor, this would place an excessive burden on them, forcing them to focus on character-level problems rather than on big-picture issues, and resulting in decreased productivity and increased error-proneness [Horwitz, 1990; Riddle and Fairley, 2012]. Instead, developers make use of semi-automated software development and analysis tools (SDATs) to analyze potential changes, to make changes, and to catch errors [Fenton and Neil, 2000; Johnson et al., 2013].

Software developers continually receive feedback from production and operations teams at frequent stages requesting improvements in the software code base which typically involves bug fixes, addressing security and stability issues, and performance issues. Companies like Amazon and Netflix undergo maintenance and deployment thousands of times per day [Hussain et al., 2017]. Software development and maintenance is achieved using continuous integration and continuous deployment(CI/CD) strategy which involves careful inspection of the new changes being introduced [Agarwal et al., 2018].The developers use SDATs to analyze the requested improvements and to identify the potentially affected areas of the proposed changes.

Most SDATs build atop one or more models of the software, such as abstract syntax trees, which are syntactic models, or type hierarchy graphs, which are semantic models [Seidewitz, 2003; Neamtiu et al., 2005]. These models can be built statically or dynamically and function on either a coarse- or fine-level of granularity. Static approaches involve analyzing and examining the source code without executing the code, to compute the model for all possible executions [Nagappan and Ball, 2005; Emanuelsson and Nilsson, 2008; Schwartzbach, 2008]. In contrast, dynamic approaches rely on information collected during program execution to perform analysis over the exercised parts of the system [Dai et al., 2009; Kolbitsch et al., 2009; Anderson et al., 2011]. There are certain times when even a combination of both are required [Khazan et al., 2005; McCorkendale et al., 2014]. Thus, the cost of computing the model differs and depends upon the characteristics of the analysis applied. Similarly, there is a cost difference based on the granularity level of the analysis: fine-grained analysis approaches require more time as compared to coarse-grained approaches because of the additional detail that they require to model; coarse-grained approaches can generally eliminate the need for intricate and precise models but at the cost of reduced accuracy [Debray et al., 1990].

The space and time costs of building software analysis models not only depend upon the level of depth of analysis and the nature of the approach applied, but also on the software being analyzed. This means that the larger and more complex the software, the more costly it is to build the model [Dyer, 2013]. Therefore, building an SDAT model for a large software system is immoderately expensive, and even optimized versions of SDATs can require long building times and large amounts of memory. For each build of the model, there will be an interim wait time between when the software is ready for analysis and when the software has been analyzed. This is an undesirable downtime in developers' daily development activities and is always likely to increase exorbitantly as software systems grow in size and complexity.

However, often, the same piece of software needs to be analyzed anew by the same or different developers because its models are no longer present in memory. Some scenarios where re-computation is required include the following.

1. When the software analysis tool is used to analyze more than one component of one or more software systems, the previous model of the analysis is lost from memory. This requires the tool to rebuild the model again every time instead of updating.

2. Tools can crash due to faults, requiring rebuilding of the model.

3. When the system is shut down at the end of the work shift of the developer, the SDATs are also stopped and their models must be rebuilt the next time.

4. Even if the tool is not shut down, the system cleans its memory periodically and the computed model is lost; thus, there is a need to rebuild again.

While rebuilding may be straightforward and a minor inconvenience for small systems, that is not the case for large and critical ones. Traditional rebuild mechanisms simply recompute the model in its entirety, which is as costly as building it from scratch. The rebuild cost can be amortized if there is some means of storing all or part of the models out of volatile core memory and restoring them to core memory when needed again, assuming that the cost of rebuilding would be greater than the cost of storing and retrieving the models. The validity of this assumption and the degree of the savings involved will depend on (a) the details of the analysis model, (b) the size of the analysis model, and (c) the mechanism of storing and retrieving the models to/from external memory. Furthermore, while deployment is performed on high-end computing facilities, software development and maintenance operations are usually performed using desktop computers or laptops [Hannay et al., 2009; Agarwal et al., 2018].

1.2 Previous approaches to the problem

Memoization is a theoretical optimization technique where the results of functions are stored in a cache of some sort and retrieved when the same computation is repeated [Michie, 1968]. Offline memoization is the process of storing the computed results in a datastore, and reusing them in future executions to save the computation cost. Examples of datastores include:

1. *flat files*, e.g., simple text files; semi-structured files such as comma-separated value (CSV) files [Shafranovich, 2005] or character-delimited files; JavaScript Object Notation (JSON) files [Crockford, 2006];

2. *relational databases*, where the data is stored in a tabular structure of rows and columns called *relations* [Codd, 1989]; relational database management systems (RDBMSs) such as MySQL, IBM DB2, and PostgreSQL are used to store and maintain the database, utilizing the standardized high level language known as Structured Query Language (SQL) for all interactions;

3. *non-relational databases* allow the storage, retrieval and maintenance of the data without a tabular structure and SQL language; thus, they are now commonly known as NoSQL databases [Han et al., 2011] (Redis[1] and MongoDB[2] are examples);

4. *graph databases* are also NoSQL databases where the data is modelled and stored as graphs, and the maintenance of such databases allows flexibility in the schema [Angles and Gutierrez, 2008]; Neo4j is a graph database management system which uses the Cypher query language[3];

5. *Cloud storage* allows the storage of files and data at a remote location, which could be accessed via the internet or a private network connection. Google Cloud datastore is a NoSQL database which allows the user to syndicate the data on cloud by SQL-like queries.[4]

[1] Redis Support, `https://redis.io/topics/introduction` [2021/06/29]
[2] MongoDB Support. `https://docs.mongodb.com` [2021/06/29]
[3] Neo4j Support. `https://neo4j.com/docs/` [2021/06/29]
[4] Google Cloud Datastore. `https://cloud.google.com/datastore` [2021/06/29]

Object *serialization* is technique that enables the conversion of the in-memory representation of a structured object into a platform-independent stream of bytes [Waldo, 1998; Philippsen et al., 2000]. This stream can be written to a file for storage or transmitted over a network for communication purposes. The reverse action of reconstructing the same object back using the stream of bytes in the same or different environment is called *deserialization*.

Prior work on such storage mechanisms provides us little evidence to leverage in deciding the best approach for serialization and deserialization in a concrete system. We can summarize the issues with such work as follows.

1. The dataset to evaluate and compare relational database and graph database is randomly generated which does not accurately represent the nature of real world software.
2. The comparison of storage mechanism libraries has been performed on small software systems. Small software systems in our context do not need any storage mechanism since rebuilding a model for them is already feasible. The research results must scale up to industrial applications for them to be useful [McDermid and Bennett, 1999].
3. Few studies implement and integrate storage mechanisms on a tool/application to compare the performance, rather than providing just a generic descriptive case study of the storage mechanism. While this helps to understand the performance better, the data to be stored/retrieved in these study is not large enough to inform our scenario.
4. Database libraries have been compared, however, their performance along with data access technology has not been taken into account. When software is integrated with a database, its performance to store and access the information also adds to the overall execution cost.
5. There is a lack of guidelines about how to make context-specific decisions about what approach to use. For example, in a real scenario, we need to consider interaction with a garbage collector, process priority, and the presence of other loads on the machine. Without such considerations, the validity of the data in such experiments is of questionable generality.

So the results tell us little as to which of the possible storage mechanisms would be best to use in a specific context, nor what preliminary evaluation to perform to restrict the large range of possibilities.

1.3 Our solution

We start by performing a case study which aims to identify the most suitable offline storage technology which could aid SDATs in minimizing the long interim time to rebuild the model. To reduce the scope of this issue to a more manageable level, we consider the problem of change propagation. Change propagation is a technique used to re-establish consistency to a system after a change has been made within the source code [Han, 1997]. Our study was conducted in the following three phases:

Phase I. We generated Barabasi–Albert graphs, which are random, scale-free, and follow the preferential attachment model [Farkas et al., 2001]. These graphs were used to investigate the growth behaviour as well as scalability of dependence graphs used for change propagation. Dependency graphs, which lie at the core of the change propagation model, are a representation of data- or control-flow between the entities of a software program. We store these simulated graphs in a relational database and graph database and performed the relevant change propagation activities which includes adding and deleting a node or edge and updating an existing dependency.

Phase II. In this phase of our study, we use ModCP [Men, 2018], as a concrete implementation to represent change propagation tools, on six open source real world software systems of different sizes and complexity as the software to be analyzed. The baseline configuration considers the extant version of ModCP in which no memoization takes place. For comparison, PostgreSQL and Neo4j are used as the underlying database technologies for memoization. However, the rebuild performance does not improve and in fact, worsens drastically as compared to the baseline.

Phase III. Finally, we conduct a study on four object serialization technologies, to capture and store data derived from 111 classes and 311 fields and properties in four different storage formats, which later can be used to reload the model when a rebuild would otherwise be required. The goal is to evaluate these serialization technologies in terms of: (a) speed, (b) schema evolution support, (c) introduction effort and maintainability, (d) documentation, (e) benign learning curve, and (f) platform- and language-framework independence.

1.4 Thesis statement

The thesis of this dissertation is that, in general, the cost of storing and reloading the models of ModCP can be much lower than rebuilding them, and that, specifically, the use of data-contract serialization obtains the best speed-up of the available options, with a twenty-fold decrease in the relative time required.

1.5 Thesis outline

The rest of this thesis is organized as follows: Chapter 2 contains motivational examples to illustrate the need for research into faster rebuilding and analysis in existing SDATs. In Chapter 3, we present potentially beneficial data storage technologies for SDATs, discuss our approach to reduce the computation cost and prior work on the performance and scalability of the discussed technologies. We evaluate these data storage technologies in the two subsequent chapters. Chapter 4 is a comparative study of select candidate database technologies to store and maintain the analysis graphs in a real-world software system. In Chapter 5, we discuss the advantages of adding database technology support to SDATs. In Chapter 6, we discuss and describe the favourable implementation of object serialization techniques in SDATs and explore the about past research on the performance and scalability of the serialization technologies. We evaluate these object serialization techniques in two chapters, where Chapter 7 is a comparative evaluation of the selected candidate object serialization techniques in ModCP and Chapter 8 is a focused evaluation of the selected object serialization technique from Study-3. In Chapter 9, we discuss the implications of our work and the threats to validity. Chapter 10 concludes the thesis.

Chapter 2

Motivation

This chapter provides a brief background on what software development and analysis tools are and why they are used by software developers. Then, we discuss the change propagation tool ModCP, which we use in our study as a representative of such tools. We demonstrate ModCP in the layout of the Model–View–Controller (MVC) architectural style and discuss scenarios where the current process of rebuilding models from scratch becomes more than an inconvenience in developers' daily development activities.

2.1 Software Development and Analysis Tools

Software development and analysis tools (SDATs) are used by software developers to create, modify, and maintain a program. Any software, once developed, still goes through a series of maintenance activities which may arise either due to issues raised by end-users, or due to the changing needs of the organization itself. Software maintenance is defined as modifications made after deployment. Software maintenance is considered as the most costly phase of a software lifecycle, requiring 70–85% of the total cost.

Just like much other software, SDATs interact with software developers through a view called a user interface. This view can consist of a graphical user interface (GUI), command line interface, or application program interface (API). The software developer interacts with the view and the view is responsible to propagate their requests to the control logic of the software. The controller examines the nature of the request and performs the operation required to fulfill it by using information stored in the core model. The controller is also responsible to initiate any modifications to the model itself.

In our work, we use ModCP to represent SDATs to identify and address the problem of long interim time involved in unnecessary re-computation.

ModCP [Men, 2018] is a change propagation prototype tool built in collaboration between the Laboratory for Software Modification Research and its industrial partner, Find it EZ Software Corporation. The tool functions on coarse- and fine-grained analyses to perform change propagation in any Java program at the statement level. Change propagation is an activity of identifying the impact of a change introduced in a program. It is used to perform risk/cost analysis and to ensure consistency of workflow in the program.

2.2 The MVC architecture of ModCP

The underlying model of ModCP comprises data from three types of graphs: (1) a class hierarchy graph (CHG), (2) an enhanced call graph (ECG), and (3) a set of modularized program dependence graphs (mPDGs). The CHG provides the inheritance information between classes and interfaces; each node in the CHG represents a class or an interface and an edge exists between two nodes when one is derived from another. The ECG represents the call chain of methods obtained from the source code being analyzed; it is also global in nature and relies on the resolution of method calls to their definition in order to be built and updated when a change is made. The CHG and the ECG provide information for coarse-grained analysis and the building of mPDGs relies on these graphs. An mPDG is built for every method within the source code, providing data and control dependence between the statements within a method through two sub-graphs: (1) the data dependence graph and (2) the control dependence graph.

The prototype tool ModCP has a simple GUI which allows the user to upload a Java project or a single-file Java program. It offers a code editor window along with a tree view of project directory. When the user selects the seed for where the change needs to be propagated, the results

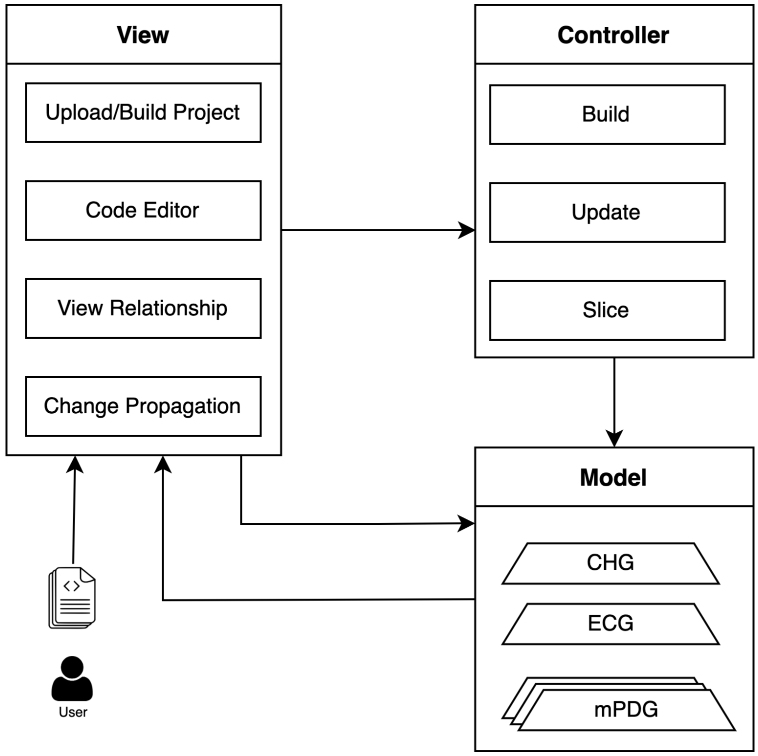


Figure 2.1: Model–View–Controller architecture of ModCP.

are displayed as interactive graphs in a viewer tab. A code editor can be used once a decision is made to move forward with desired/required changes. Following this, the user can save the newly changed code and the model is updated.

Figure 2.1 gives an illustration of the architecture of ModCP. The software developer uploads the project code for analysis using the “Upload/Build” view. The view forwards the code file to the controller, which is responsible to build the model. The source code files are uploaded and a lexer generates a parse tree which is used by a parser to build an abstract syntax tree (AST) for the given code. The AST is used to build the graphs that make up the model. Once the model is built, the view is changed and the user can either perform change propagation or make changes using the “Code Editor” view. The “Change Propagation” view requires the developer to select a statement where a change is suggested, the level of granularity at which the analysis is required (coarse or fine), a seed node (which can be a local variable or a field), and a distance for analysis depth.

Once the user provides these parameters to perform change propagation, the view sends the user request to the “Slice” controller where the ModCP’s slice algorithm is performed. The slice algorithm retrieves the model data which is stored in the volatile memory of the system. At last, the results are passed to the "View Relationship" view, where potentially impacted nodes and edges are visualized as an interactive graph. The user can navigate these graphs, increasing or decreasing the depth, to better understand the cost and risk involved in the maintenance activity. Once the course of action is finalized, the changes can be implemented using a "Code Editor" view. The modified source code files are notified to the "Update" controller by the view, which is responsible to update the model data accordingly. Since mPDGs are modular in nature, they are updated efficiently as only the affected ones are changed. The rest of the graphs are updated based on the affected set gathered by the viewer. The CHG is updated first since the ECG depends on it. The CHG is updated in the following sequence: the nodes of the deleted classes are removed first; the nodes of renamed classes are replaced with new name labels; if the parent for a class is changed, the outgoing edges are updated accordingly; a new node is created for every new class added. The ECG update process is intricate but basically follows the same pattern as the CHG update.

Since all the graphs are built when the source code is loaded into the tool and slicing is done on demand, this approach makes it faster to update the model when some change is made; however, every time the tool is shut down the user must wait again (for more than half an hour for larger input projects) for the graphs to be built before proceeding to propagate a change. Therefore, even though the slicing and updating of the graphs is faster than other approaches as suggested by their evaluation results Men [2018], the tool requires storage support so that the wait time may be decreased. The goal is to only build the model once, and then if the tool is shut down, the model may be restored from the storage and changes can be made to the source code without waiting.

2.3 Motivating Example

To understand the existing drawbacks and need for memoization in SDATs, let us consider a scenario, illustrated in Figure 2.2, where ModCP is being used as an SDAT to assist the software engineers at ABC Corporation in maintenance work. ABC Corporation is a game development company which currently offers three famous Android games: X, Y, and Z. Due to the competitive and ever changing gaming world, the developers at ABC Corporation perform a biweekly in-app update which includes weekly missions and challenges for the game users. Apart from the biweekly in-app updates, the game version becomes obsolete in every three months for the users. Every three months, the season theme and rewards are changed for the game and the users who are using the older version cannot play together with other players using the newer version. The reason behind this is that the newer version has different nodes and events which are incompatible with the older version.

ModCP is used by the developers to assist them in assessing the proposed change and identifying the potentially impacted code blocks/statements when adding/removing/modifying an existing functionality. Just like every other workplace, the developers at ABC Corporation are not permanent employees of the company and they tend to change over time. So the new developers have little to no clue about the existing code base and they mostly rely on ModCP to examine the potential regions which may require their attention to modify the code blocks. The biweekly changes

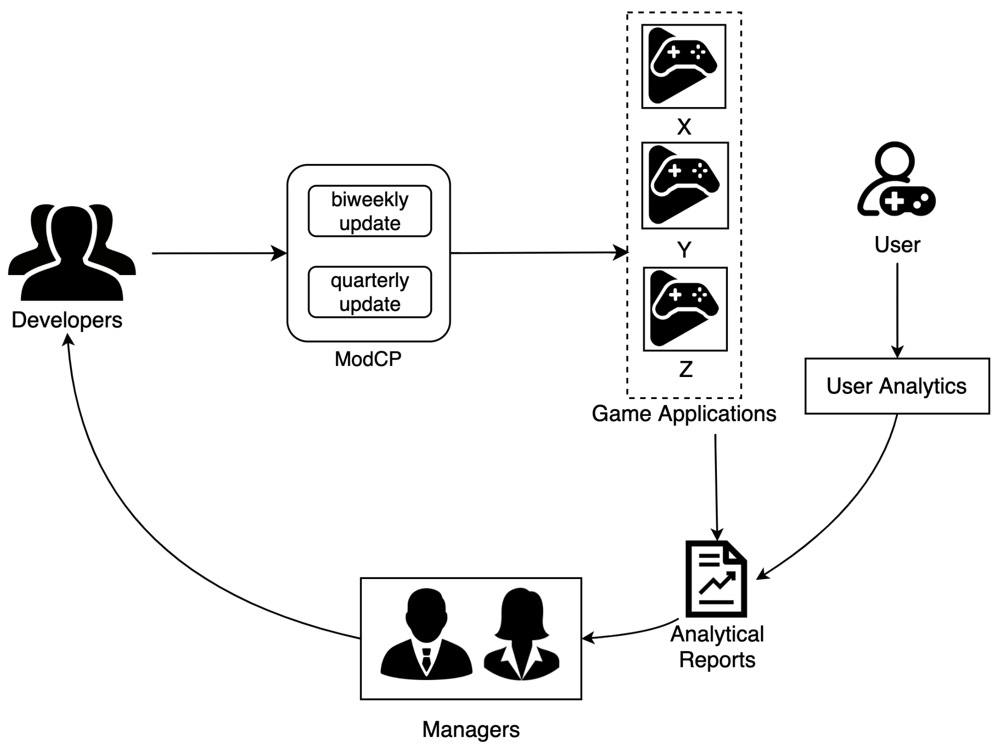


Figure 2.2: Implementation of ModCP and its working in the game applications of ABC Corporation.

introduced in the games are minor, but a good quality control job is very crucial part of this job. A minor bug can crash the whole game which might require urgent fixes and re-downloading of the new version, which clashes with the whole idea of in-app update. The biweekly in-app update, therefore, goes through rigorous testing in a brief time frame and thus is costly when compared to the offline update.

2.3.1 Scenario 1.

John Doe, a new developer at ABC Corporation, uses ModCP to seek assistance in performing maintenance work on the game X.

John is responsible to address the bug reports and analytic reports provided by the managers and end users, in the upcoming biweekly in-app update. John starts his work shift by starting to work on game X. He uploads the source code of the game into ModCP for change propagation assistance. ModCP, as X's source code is uploaded, builds the analysis model. During this stage, John cannot do much since ModCP is in the loading stage. It took approximately 30 minutes until John could actually make use of the tool's features, like locate potential impacted regions and cost analysis. John continued his work for the rest of his shift. The next day, John repeats the same process and waits for another thirty minutes out of his productive work.

Now, that John gets accustomed to the new workplace, he tries to maximize his output by performing the loading process before leaving for a break. But, John still finds this whole repetitive process inconveniently slow. When the workload is high (and time is of the essence), and ModCP is on the model building stage, John often proceeds without the cost and risk analysis, sometimes introducing even worse bugs that require even longer to diagnose and repair than if he had worked in the standard way.

2.3.2 Scenario 2.

John Doe adds new functionality in game Y and modifies existing functionality in game Z, using ModCP.

John, just like other employees of the company, has to frequently work on more than one game to keep up with the schedule. In such instances, John has to load the codebases of games Y and Z simultaneously several times a day to complete the work. In these kind of scenarios, John tends to lose more than hour of his shift and has to even work past his shift hours.

2.3.3 Scenario 3.

John Doe returns to work after the weekend to resume maintenance work on game X.

John, over his time at the new workplace, found a work around for evading the loading phase interim time by not signing off at the end of the shift and by just locking his system. But, when the system remains idle for too long, the memory refreshes on its own periodically and thus loses the model which was loaded in ModCP on the volatile memory. Another reason that the model gets lost even when the system is not signed off is in the event of automatic updates that occur overnight as per company policy.

2.4 Summary

In this chapter, we defined and discussed various software development and analysis tools. We selected ModCP, a change propagation tool, as an implementation of such tools for our study. We described ModCP using the Model–View–Controller architectural style and briefly talked about its working. We provided an example to point out various scenarios where unnecessary re-computation is performed in ModCP. We concluded that the building of analysis models for large real software systems can occupy significant time during which further work on these systems would be highly risky. This is an inconvenience for software developers and even after trying any workarounds, it still remains an issue. The interim time gets worse when working on multiple projects at the same time as the process of building has to be repeated since it is stored in volatile memory. ModCP requires a storage support where the model data could be easily stored and retrieved by the controller, thus avoiding the re-computation in the scenarios described here.

Chapter 3

Database Technology to Address the Problem

In this chapter, we consider and compare various available database technologies for possible integration in SDATs to decrease the re-computation cost of the model. We consider and explore the two major datastore types—relational databases and non-relational databases; we discuss our attempts at understanding the data to be stored and its growth behaviour in the database. We also explain the process used to integrate the technology implementation (like PostgreSQL, MySQL) in ModCP to identify which database technology could be integrated to maximize efficiency in rebuilding the model,

3.1 Database technology and its potential benefit in SDATs

During the maintenance phase, the re-computation cost of the same piece of software anew is rather expensive as it comes with an undesirably-long interim wait time between when the software is changed and when the software has been analyzed. These long wait times could be reduced if they are stored in a data-store and loaded in the memory when needed, rather than re-computing the model from scratch. Relational databases and non-relational databases are two such data-store technologies that could be integrated in SDATs to store the model(s). Relational database management systems (RDBMSs) are based on the relational model in which data is managed in a structured manner using relations (tables) and these relations may possess relationships among themselves. RDBMSs use structured query language (SQL), a programming language which is used to communicate and handle structured data, consisting of two major components: the data definition language, to define/create the database, schema, users, etc.; and the data manipulation language, to insert, modify, and delete the data. Non-relational databases, also widely known as NoSQL databases, are used to store data which is not modelled as tables. Unlike relational databases where SQL is used to manage the database, there is no standard programming language to handle non-relational databases.

The never ending debate about which database technology is better has led to several studies in the research community where both types of database technologies were implemented on different scales using a range of database technology representations; the results are not straightforward. The mixed findings indicate that the decision to choose between them should be based on the needs of the client. The relational database keeps the data very well organized; however, the rigid schema is also a major limitation. Non-relational databases, as the name states, allow flexible schema and scalability in managing huge volume of data. With the increase in the size of data and its complexity in the relational databases, the query write-up and its processing time gets high too, whereas NoSQL offers simpler and easy querying of database. However, the absence of standardization of a query language in NoSQL results in a long integration time due to a steep learning and training curve.

The model of ModCP is lost when the tool is closed or another software is analyzed. The model, as discussed in Chapter 2, could be stored in a datastore and reloaded when rebuilding of the model would otherwise be required. We consider both types of database technologies to identify the best suitable storage platform for ModCP.

1. **MySQL**. MySQL is one of the most used, open source, traditional relational databases that is considered reliable due to high availability and around the clock up-time. It provides uninterrupted access by enabling instant fail-over through a wide variety of cluster servers and

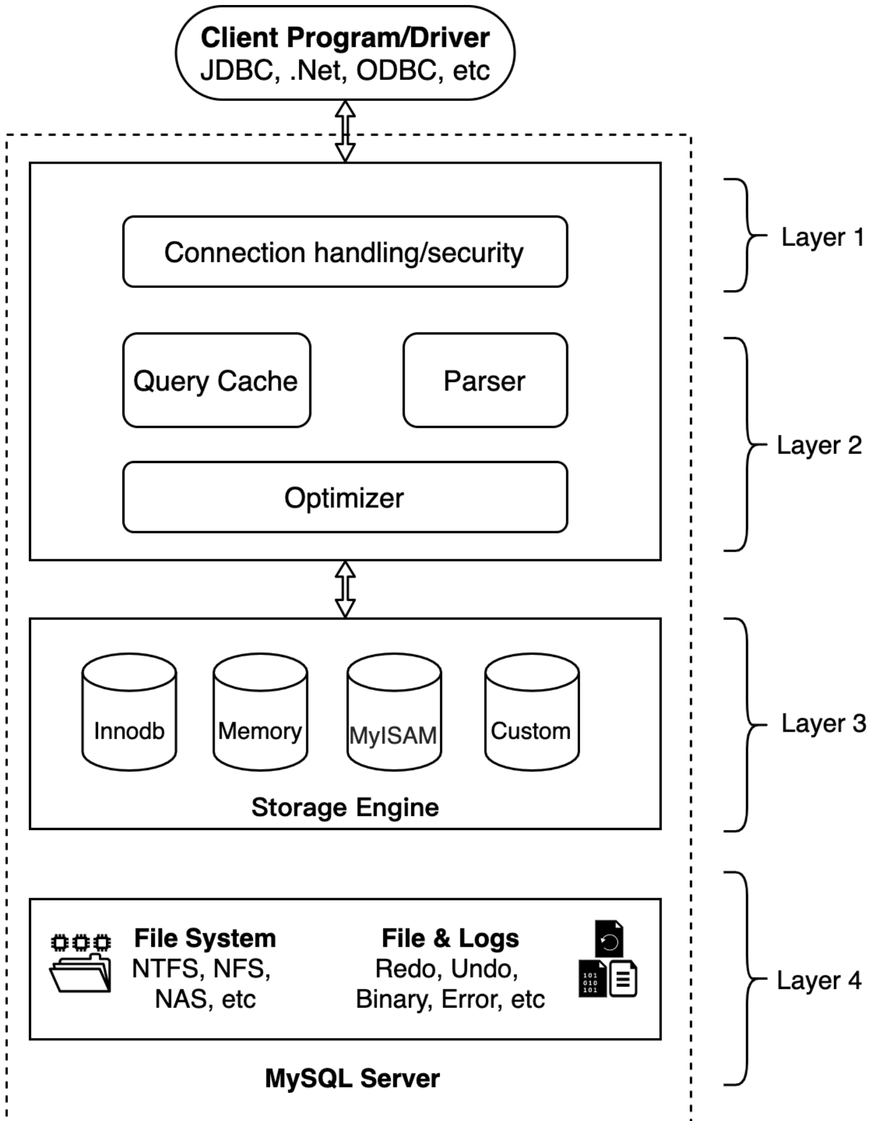


Figure 3.1: MySQL architecture.

master-slave replication configurations. Any high-speed processing system can use MySQL as it is designed to process millions of queries and thousands of transactions while ensuring rich functionalities, with great technical support and maintenance.

The MySQL database software is a client/server system that consists of a multi-threaded SQL server that supports different back ends, many different client programs and libraries, administrative tools, and several application programming interfaces (APIs). An embedded multi-threaded library is also available that can be linked into an application to get a lighter and faster standalone product. MySQL has a pluggable storage engine architecture, shown in

Figure 3.1, which allows the selection of a specialized storage engine for a particular application need while encapsulating specific application coding requirements in case application changes require changes in the underlying storage engine. The application programmer and the database administrator interact with the MySQL database through Connector APIs and service layers atop the Storage Engine. The top layer, Layer 1, in MySQL Server, shown as dotted rectangle, shows the services that MySQL provides for client/server networks including connection handling, authentication, and security. Layer 2, which is the processing center of MySQL, includes query analysis, optimization, caching, and other functions available in MySQL. The third layer consists of pluggable storage engines and the last layer is the files system where physical storage of the data is kept.

2. **PostgreSQL**. PostgreSQL is another open source RDBMS which is developed, updated, maintained, and fixed by a devoted group of volunteer developers around the world. PostgreSQL, similar to MySQL, uses a client/server model, as shown in Figure 3.2, supporting the standard SQL as the querying language. PostgreSQL is an object-relational programming language (ORDBMS), where as MySQL is a pure RDBMS [Douglas, 2003]. It serves as a bridge between object-oriented programming and relational/procedural programming. This translates to supporting complex data structures by allowing the developers to define objects and table inheritance. An ORDBMS is especially useful when dealing with data that is not compatible with a strictly relational model. PostgreSQL can perform complicated read/write operations, while using data that requires validation. It supports NoSQL and a large variety of data types including JSON, hstore, and XML. Original data types can be defined and custom functions can also be set up as well.

   PostgreSQL is designed for extra-large database management making it an attractive choice for our use cases. Figure 3.2 shows the different components of its architecture. A server process called postgres, shown in stacked oval shape, manages the database files, accepts connections to the database from client applications, and handles the client requests. On the other hand, client applications can be a text-oriented tool, a graphical application, a web server, or

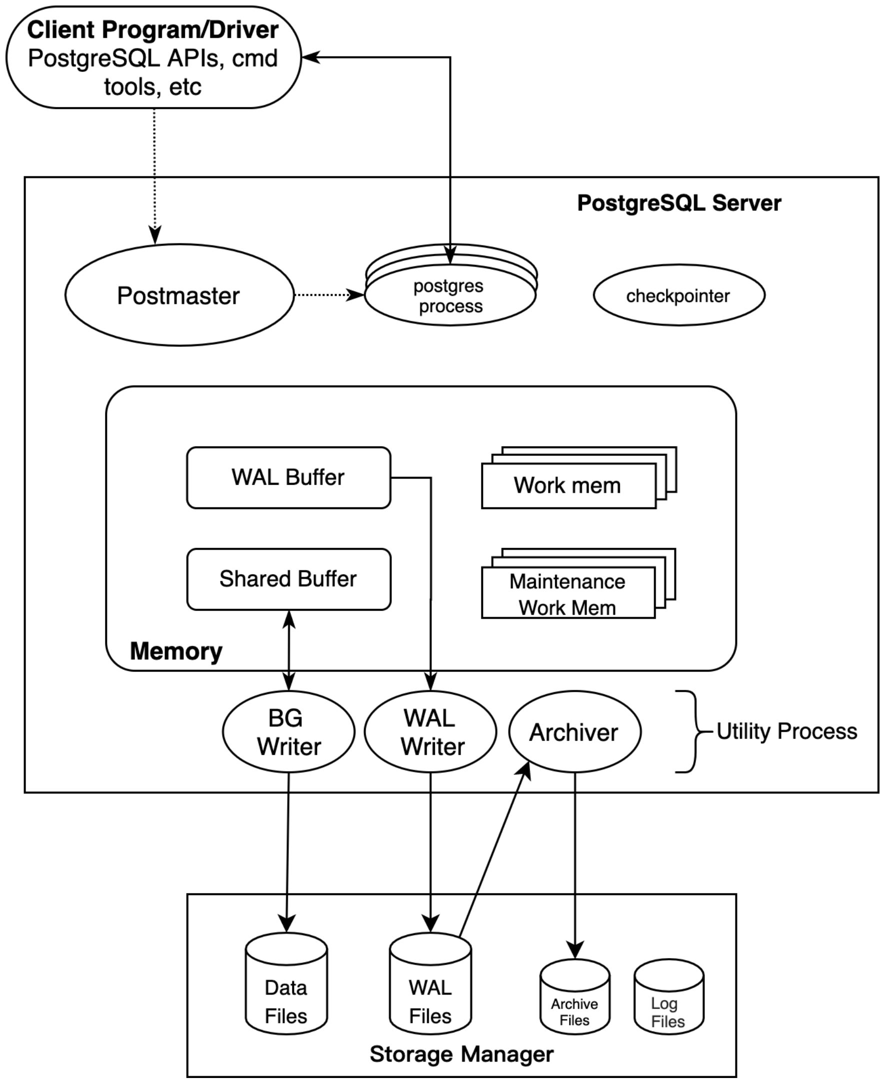


Figure 3.2: PostgreSQL architecture.

a specialized database maintenance tool. The physical structure of PostgreSQL consists of the following components: (1) shared memory, (2) background processes, and (3) data directory structure/data files. Write-Ahead Logging (WAL), a mechanism for ensuring data integrity, refers to the concept where the changes to database tables are only processed once those changes have been logged [Shaik and Vallarapu, 2018]. WAL buffers temporarily store changes to the database, work memory is used by internal sort operations and hash tables, and maintenance work memory is used by maintenance operations like vacuum, create and alter table, and add foreign key operations. The background writer handles the job of writing to disk specific dirty buffers, whereas the job of the checkpointer process is to write the change to the data file.

3. **Neo4j**. Relational databases depend on rigid schema and do not make it easy to add new relationships between objects. Implementing such problems in relational databases involves high cost due to a large number of join operations which can expensive. This led to the birth of non-relational databases, or NoSQL (the NoSQL name was coined because they do not use the SQL language) "Not Only SQL" databases as a response to web data. NoSQL databases do not tend to have a standard language. They offer a means to store and extract unstructured data quickly which is managed for different types of analyses.

   NoSQL databases can generally store data in four ways:

   - *Key–value store*: a data storage system designed for storage, retrieval, and management of"associative arrays" (e.g., DynamoDB, Redis, Voldemort) [Kepner et al., 2016].
   - *Wide-column store*: uses tables, rows, and columns, to store data but unlike RDBMS, names and formats of the columns can cross rows within the same table (e.g., Cassandra, HBase) [Sharma and Dave, 2012].
   - *Documents store*: designed for storage, retrieval, and management of semi-structured data (e.g., MongoDB, CouchDB, DynamoDB) [Guimaraes et al., 2015].
   - *Graph store*: stores a collection of relationships in the form of nodes and edges (e.g., Neo4j, InfiniteGraph) [Miller, 2013].

   Graph databases, a type of NoSQL, are databases that store data in the form of a graph and that are optimized in analyzing relationships between data rather than the data itself. The graph consists of nodes and edges, where nodes act as the objects and edges act as the relationship between the objects. Nodes and edges have properties that depict their specific attributes. In Neo4j, "relationships are first-class citizens of the graph data model", unlike relational databases [1]. Graph databases were designed to work with huge datasets. It makes use of index-free adjacency consisting of a direct pointer to the adjacent node. This makes graph

[1]Factors Driving the Graph Database Explosion. https://www.datanami.com/2016/08/19/5-factors-driving-graph-database-explosion/ [2021/06/29], https://neo4j.com/news/5-factors-driving-graph-database-explosion/ [2021/06/29]

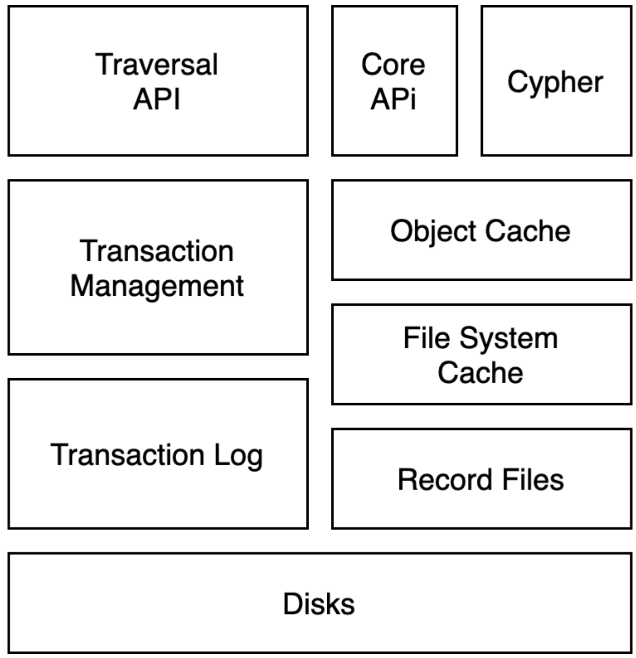


Figure 3.3: High-level architecture of Neo4j.

databases perfect for search queries as these can traverse efficiently and fast. Some of the best known graph databases are: Infogrid, HypergraphDB, Jena, DEx, FlockdB, and Neo4j. Out of these only Neo4j is discussed in the remaining part of this section.

Neo4j is graph database system software written in Java and using the Cypher query language. Neo4j stores various parts of the graph structure in a set of files known as *store files*. Stored files are generally categorized by their record type; therefore, separate files exist for nodes, relationships, properties, labels, etc. Neo4j can be deployed on multiple systems. It comprises two parts: (1) a client that sends commands to the server and (2) a server that processes the commands and responds with the results to the client.

Each layer of the high-level architecture of Neo4j (shown in Figure 3.3)—from the Cypher query language to the files on disk—is meant for storing graph data in the most optimal way. The two APIs, traversal and core, provide support for accessing the graphs; object and file caching allows for improved performance; and transaction management makes this database fully ACID compliant. Record files store various aspects of the graph structure whereas the transaction logs record all write operations in the database.

The model of ModCP is an instance of a class which is stored in memory and the underlying data are the field and its value pair. The size and complexity of the model object is therefore based on the nature of the software being analyzed. The model of ModCP can also be perceived as graphs. Class Hierarchy Graphs, Enhanced Call Graphs, and mPDGs: these three graphs constitute the ModCP's model. Based on these two ways of perceiving the model of ModCP, both RDBMS and graph database could be used as an implementation of data store technology.

To identify the most suitable data store technology, we conduct a study on relational and non-relational data-store to store, retrieve and maintain the model of SDAT. By doing so, we target to understand the nature of ModCP model data and how well the database technologies scale with the increase in the size and complexity of software being analyzed.

3.2 Prior work on the performance and scalability of database technologies

In this section we review some of the major work in this area. The focus remained the identification of a database technology that could be implemented for our use-case. Existing relevant work ranged from comparison of different kinds of database technologies for generic datasets or for specific purposes. The goals of these works varied, therefore: while Vicknair et al. [2010] focused on finding the differences between a traditional relational database and NoSQL databases, or Fioravanti et al. [2016] wanted to find way to model the transition from a relational to a non-relational database. Moniruzzaman and Hossain [2013] claims to help users in understanding the merits and demerits of NoSQL databases to support applications processing huge volumes of data. However, the author considers the volume of data as a factor to choose NoSQL database approaches and does not take scalability into consideration. Also, the comparison is based on the characteristics of different NoSQL databases (key–value, document based, wide-column) rather than storing datasets of distinct size and complexity, which does not aid us in our selection scenario. This work focuses on evaluation based on attributes of the database technologies such as design, integrity, indexing, and distribution; no real studies are performed to test their performance. Another factor that they considered is the popularity for these technologies in the job market which may indicate where the demand lies but does not necessarily help in deciding which technology we can use for different requirements.

Vicknair et al. [2010] acknowledged the increase in emergence of NoSQL databases and performed a comparative study to understand the differences. The author(s) chose MySQL as to represent relational databases because it is open source yet commercially recognized and Neo4j to represent graph databases which is fairly new. They researched and recognized some specific areas where they believe using non traditional NoSQL databases might come up as an improved alternative. Besides comparing the space and time taken by both different database type platforms, they also compared them on aspects like customer support in terms of documentation and ease of use. They created datasets of different size to address scalability. Two types of datasets were created: character database and integer database. Both of these datasets had a range of available size from 1,000 to 1,000,000 nodes. The space required to store these datasets on the databases and time taken to run a set of queries was recorded. The results demonstrated that Neo4j performed better than MySQL in structural queries. The idea to store graphs in relational as well as graph databases was taken because of the nature of data and information around us exhibits graph patterns. Social media networks, customer and commodity interaction, and even dependence between software modules in software development and analysis tools are graphs. However, the major setback of this study is that the datasets were generated with random values. Real world graphs are not random and they exhibit power law distributions, being scale-free. Another difference from our requirement is that the data was stored in the database directly and then compared. In our use case, the database is supposed to be integrated with a software or tool. In this scenario, not only does database technology have a cost, but also the cost of data access technology which is responsible for the communication between the application and database is also to be included.

Batra and Tyagi [2012] also compared relational and graph database technologies: MySQL and Neo4j. The evaluation parameters of their study included level of support/maturity, flexibility, and security. MySQL was queried using PHP and Neo4j was queried using the Cypher query language. Three predefined queries were used for the evaluation and the time taken to execute these queries was recorded. Their results show that the retrieval time for MySQL increases drastically based on the number of objects retrieved whereas only a slow increase in time taken is seen for Neo4j. One of the flaws in this study is that only data retrieval is considered as a factor for evaluation; addition/deletion of nodes in graph databases or any other operation is not considered. In addition, simple example objects with simple relationships have been used for evaluation which do not represent real-world systems where more complex relationships exist and require a more thorough evaluation.

With sensitivity to the characteristics of the dataset in mind, Fioravanti et al. [2016] performed an evaluation of MySQL+Hibernate, Neo4j, and MongoDB. Their work centres around changing of the storage schema from a relational + ORM persistence stack to a NoSQL solution. They describe their approach of modelling a reflection-based architecture over Neo4j and MongoDB. Neo4j is a graph-oriented database while MongoDB is document-oriented. For their test data, they use two different kinds of data-sets: (1) a real dataset obtained from 13,000 clinical examinations, which includes 243+110+99 fields, organized in a graph with a depth (i.e., the maximum distance from the root node) of 8, and (2) a synthetic dataset which contains generated examinations organized in a full binary tree structure in which the depth ranges from 2 to 8 and contains 100 examinations per depth. Only the read operation has been considered in this study which leaves room for further evaluations. The time taken for the read-only operations for a single examination by each implementation is recorded which includes retrieval of the examination. Their results indicate that both NoSQL implementations show improvement in performance metrics. Neo4j reduced the retrieval time by approximately 1.5 times, and MongoDB reduces it by more than 33 times. They also state that their implementation of NoSQL technologies shows a better scalability and flexibility with Neo4j requiring less engineering investment than MongoDB for the data model conversion; however, the MongoDB implementation shows better performance. The conclusion made in this study rely solely on one operation and requires further investigation for the scalability of graph database vs relational databases.

An in-depth survey by Angles and Gutierrez [2008] went into great detail for describing the various graph database models and how they differed from the most influential database models at the time. They provided historical data which provides a very broad analysis of literature on the graph data models and query languages graph. Later, Angles [2012] performed a comparative study of available graph databases. Their measures of comparison were general features for storing and querying data as well as database modeling features such as data operation and manipulation, query languages, and integrity constraints. Based on the level of maturity of the database management system they study two types of technologies: graph databases and graph stores. The graph databases that they selected were AllegroGraph, DEX, HypergraphDB, InfiniteGraph, Neo4j, and Sones. They also compared graph store technologies including Filament, G-Store, Redis graph, and VertexDB. The database languages under consideration were the Data Definition Language, the Data Manipulation Language, and the Query Language. In addition, the graph data structures that they considered were simple graphs, hypergraphs, nested graphs, and attributed graphs. The results of this study reveal that none of the technologies studied support nested graphs while Sones supports the most number of properties for data structures (i.e. types of graphs, edge, and nodes). HypergraphDB supports the most number of data storing features whereas AllegroGraph and Sones support the most number of operation and manipulation features. DEX and InfiniteGraph support the most number of representations for representing entities, properties and relations at both instance and schema levels. In terms of support for querying mechanisms, AllegroGraph provides the most number of options with either full or partial support for all options considered. While this work provides important insight into the features of various graph database technologies, it is lacking a lot of context regarding how these technologies would perform and scale based on the size and complexity of the database. The basis of comparison was just the features of these technologies; however, our major goal aims for better performance. Given how they do not consider physical and implementation considerations, this work cannot be used to determine which technology may best suit a given use case, but may be used as a starting point in order to explore graph databases.

Since different databases technologies have different strengths, the question that Rautmare and Bhalerao [2016] looked for an answer to was whether NoSQL databases would perform better in multiple scenarios than a SQL database for a small scale internet-of-things application. The two databases that they compared were MySQL and MongoDB. To conduct their experiment they designed the hardware of a water sprinkler system by using the readings captured from a soil moisture sensor and other sensors. The readings were then stored in MySQL version 5.5 by creating a new schema and, for MongoDB, Mongo Booster 1.6.2 client was used to create the collection and documents. The study involved using varying numbers of threads and records and while the number of threads was varied from 1 to 10, the number of records varied between 1,000 and 10,000. The read process was carried out by using a SELECT query to fetch 1,000 records from 10,000 records in the database. With a varying number of threads, their results show a similar response time for SELECT query operation for both MySQL and MongoDB till the number of threads goes up to 4. However, at 10 threads, they saw an increase in the response time of MongoDB as compared to MySQL. For the INSERT operation with varying number of threads, MongoDB was faster as the number of threads increases. The SELECT query operation for varying number of records showed an increasing trend line of the response time for MongoDB whereas it sightly decreased for MySQL, on the other hand, for the INSERT query with varying number of records: a steady line of MySQL response time can be observed. On the other hand, rapid variations in MongoDB are observed. Their results show that in some scenarios, MongoDB required less response time compared to MySQL whereas MySQL responses were more stable throughout the study. As is clear, their study only considers a small scale project with the only measure of performance being the response time. Similar to Rautmare and Bhalerao [2016], Győrödi et al. [2015] also used MongoDB and MySQL to do the comparison of a NoSQL and a SQL database. A user-forum-discussion dataset was created on MySQL and MongoDB with 10k instances and 7 attributes. Basic create, read, update and delete (CRUD) operations were used to record performance. Their results show that 10,000 users were inserted into MySQL in 440 seconds, while in MongoDB, the time was just

0.29 seconds. Similarly for read, update, and delete operations, MongoDB appeared to be perform better than MySQL. The problem lies in the fact that they did not address scalability in their study, neither were any other subjective measures like security, maturity, and flexibility considered in the evaluation. The dataset in the study is nowhere near the size and complexity of real world data.

A similar study for comparing RDBMS and NoSQL databases was conducted by Hammes et al. [2014]. They looked at structured and unstructured data and implemented the data models in two databases: MongoDB and PostgreSQL. They used two Linux cloud servers by Amazon Web Services and loaded each with the same data. The evaluation process included running a script to perform Create, Read, Update, and Destroy operations in addition to recording the execution times for bulk read/write operations. The datasets that they performed the study on were banking data and online blog data, as the structured data and unstructured data respectively. Their results indicate that for highly structured data PostgreSQL performed significantly better in terms of execution time due to join features whereas stored functions were required for MongoDB. Moreover, for unstructured data, the time taken was 1,061 milliseconds for PostgreSQL vs. 4,784 milliseconds for MongoDB for one of the queries. MongoDB's clear advantage is the ease of use as the queries can be built into scripts. Considering the overall performance, including the bulk read/write operations where PostgreSQL performed significantly better with the increase in the number of operations, their results indicate that PostgreSQL is the better option for both structured and unstructured data in terms of execution time even though the decision to choose one over the other is still dependent on the client preferences. Their results influenced our decision to consider PostgreSQL as one of the options. The cons of their study include the lack of varying sizes and complexity in the datasets as well as the types of queries performed.

Petri [2005] also carried out a comprehensive comparative study of MySQL with another RDBM by the name of Oracle9i. The study sheds a light on the functionalities being offered by the each of these technologies and provides insight into the features of each technology along with their limitations. However, performance comparison was not performed and the scope of this study lies within the boundaries of feature comparison due to the author concluding that such comparison is not plausible based on the difference in their functionalities and capabilities (MySQL does not provide clustering whereas Oracle9i does). The author points to the performance benchmarks provided by the websites of these technologies. The paper does help in identifying whether the implementation within the database technology is possible or not and offers information regarding the features that might be needed; however, it still leaves the reader to do the job of identifying the better performer on their own. The study is limited to the relational database technologies and does not consider any NoSQL database technologies to serve as a fair comparison for our work.

3.3 Proposed approach to reduce the re-computation cost in SDATs

As stated earlier, we use ModCP to represent the SDATs in our work. We apply the approaches to minimize the re-build time of its model to demonstrate and discuss the possibilities. Below in this section, we discuss these approaches and the studies based on them to familiarize the readers with upcoming chapters.

3.3.1 Study I

In the first phase (Chapter 4), we artificially generated datasets simulating the SDAT model, stored these with candidate database technologies, and managed them. We define and implement the use cases to identify potentially ideal database technology to store and manage the model data in the most cost efficient manner and reduce the re-building cost of model in SDAT.

3.3.2 Study II

In the second phase (Chapter 5), we implemented and integrated the most suitable database technology of each type, relational and non-relational, identified from Study I (Section 4) into the change propagation tool ModCP.

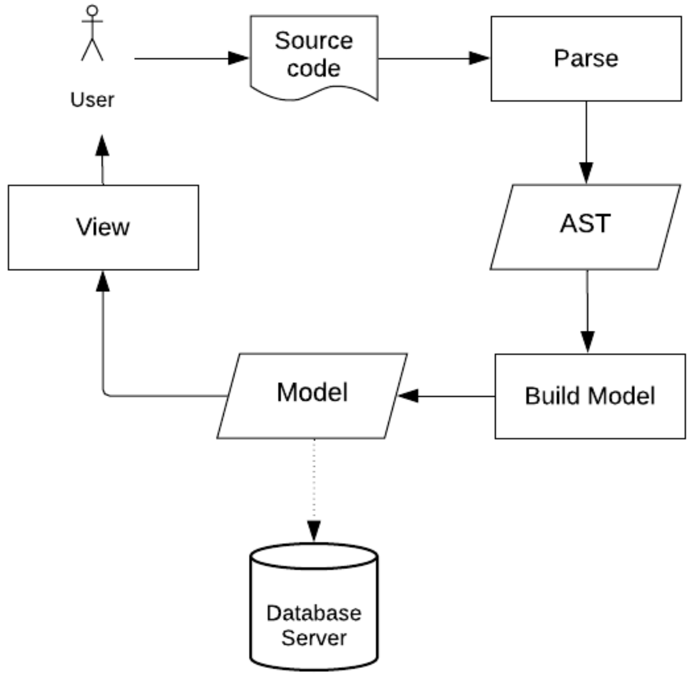


Figure 3.4: High-level structure of ModCP.

The computed model of ModCP can be encoded and stored on a database technology using a data access technology. The implementation design is described in Figure 3.4. The end user uploads the software codebase for which the analysis is required into the ModCP. The input codebase is parsed and its model is computed by generating the CHG, ECG, and mPDGs for the software. These graphs are then encoded and stored on the database technology by using the .NET database connector library. The computed model which is present in the memory is used to load the view of the ModCP tool, so the user can perform the required analysis. However, when the model data is not present in memory, the model is re-computed by retrieving the data stored in the database technology. Figure 3.5 demonstrates how ModCP can leverage the model data stored on the database and recompute it without actually parsing the codebase. ModCP establishes a connection with the database technology by the means of ADO.NET and database library which contains the necessary procedures to run the database management queries using .NET. Once the model data is retrieved from the database, the model object is re-instated by feeding the values to the attributes. This retrieving and feeding process from the database enables the ModCP tool's view to be used again by the end user similarly as if the model was rebuilt in the traditional way, that is parsing and building from scratch.

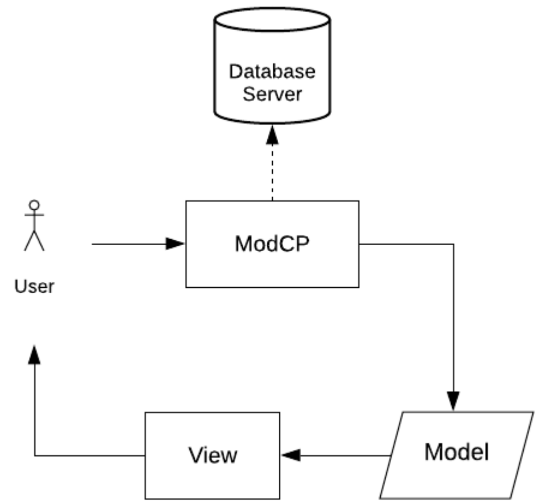


Figure 3.5: The idea of using ModCP via retrieved models.

3.4 Summary

In this chapter, we provided the introductory information about certain database technologies, the mechanisms behind their operation, and how they are considered beneficial to use as the storage mechanisms in SDATs. We also overviewed the two studies conducted to evaluate these technologies.

Chapter 4

Study I: Database Technologies in Isolation

In this chapter, we compare the database technologies we have identified as pertinent for their potential to reduce re-computation costs in SDATs. The past work in this area is insufficient to determine an optimal datastore: they lack real-world sized datasets, they only consider quantitative aspects, and they do not address many factors that would play a critical role when database technology is integrated with SDATs/software. We consider relational datastores (MySQL, PostgreSQL), semi-structured datastores (Python-CSV), and non-relational datastores (Neo4j) in our study as the candidate database technologies. We define a set of use cases that would be pertinent to SDATs and evaluate the performance of the database technologies in conducting these use cases for graphs of varying sizes. The purpose of this study is to address the following research question:

> **RQ1**: *How do different database technologies perform in comparison to one another in executing store, delete, and change operations over analysis graphs in a real-world software system?*

We find that Neo4j, a non-relational database, and PostgreSQL, a relational database, show the most promise for inexpensive, scalable datastore technology in realizing our use cases. It is important to emphasize that this study does not consider the costs of reading and writing data between the in-memory representation of the datastores and the SDAT; considering the additional costs incurred at that stage, as mediated by the relevant database connector technology, will be the subject of Study II in the subsequent chapter.

We describe the measures of performance that we utilize in Section 4.1. Section 4.2 describes the graphs that we generate as the experimental data for this study. The study setup is explained in Section 4.3. We present the results and our analysis thereof in Section 4.4. Remaining points for discussion, relative to this chapter, are given in Section 4.5. Section 4.6 presents our conclusions in response to the research question.

4.1 Performance measures

We utilize objective and subjective measures to evaluate in this study. Objective measures include the complexity of the model graphs (number of nodes and of edges), the time taken to process a query, and the space required to store the model data on the candidate database technology. The subjective measures include the difficulty of installation of a database technology, maturity level, ease of use, and the maintenance required; the subjective measures were obtained from the official documentation, open source discussions, and the published literature.

4.2 Experimental dataset

4.2.1 Mathematical preliminaries

A *graph* G is a pair (V,E), where V is a set of *vertices* (also known as *nodes*) and E is a set of *edges*, such that $E \subset V \times V$ and $\forall e \in E, e = (v_i, v_j) \Rightarrow v_i \neq v_j$; we concern ourselves with only undirected graphs in this study, without self-loops, and where any pair of nodes possess at most one edge between them. For convenience, we define $n = |V|$ and $m = |E|$ relative to whichever graph is under consideration in the given context.

An edge $e = (v_i, v_j)$ is said to be *adjacent* to v_i and to v_j; the set of edges adjacent to vertex v is the set of all edges $e \in E, e = (v_i, v_j)$ such that $v = v_i$ or $v = v_j$; the *degree* of vertex v—represented as $\deg(v)$—is the cardinality of the set of edges adjacent to it. The minimum degree of a graph

G—represented as $\deg_{\min}(G)$—is the smallest degree of all vertices in V; the maximum degree of a graph G—represented as $\deg_{\max}(G)$—is the largest degree of all vertices in V. The average degree of a graph G is given by

$$\overline{\deg}(G) = \frac{1}{n}\sum_{i=1}^{n} \deg(v_i). \tag{4.1}$$

The maximum number of edges in a graph G is given by

$$m_{\max}(G) = \frac{n(n-1)}{2}. \tag{4.2}$$

The density d of a graph G is given by

$$d(G) = \frac{m}{m_{\max}}. \tag{4.3}$$

4.2.2 Generated graphs

We generated a total of nine graphs G_1–G_9, as detailed in Table 4.1. These are scale-free graphs, generated via the Barabási—Albert model [Barabási and Albert, 1999] following the linear preferential attachment rule—also known as “the rich become richer”; the implementation used to generate them (in Python) is shown in Listing 4.1 utilizing the *networx* library that supports such generation [Wan et al., 2017]. This model represents a random dynamic graph grown from a small “seed” graph by an indefinitely repeated addition of a new vertex with m edges. The free ends of the edges of each vertex are preferentially connected to vertices that are already rich in connections. The probability p_i of connecting an edge with the vertex v_i is proportional to the local degree of connectivity k_i of v_i [Hagberg et al., 2008; Schult and Swart, 2008]:

$$p_i = \frac{k_i}{\sum_{j=1}^{n} k_j}. \tag{4.4}$$

As per Diestel [2017], *sparse* graphs are those whose number of edges is about linear in their vertices. Similarly, *dense* graphs are those in which the number of edges is close to the maximal number of edges [Black, 1998]. Figure 4.1 shows the visualization of sparse, “semi-dense” (i.e.,

Table 4.1: Utilized scale-free graphs generated via the Barabási—Albert model.

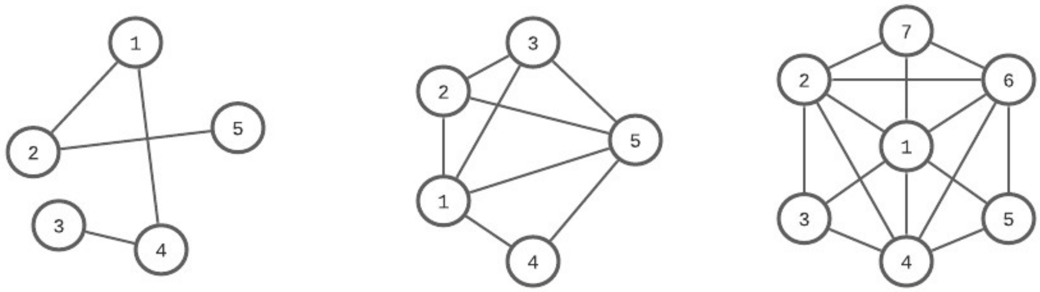
	2% Density			**10% Density**			**25% Density**		
	G_1	G_2	G_3	G_4	G_5	G_6	G_7	G_8	G_9
n	100	1,000	10,000	100	1,000	10,000	100	1,000	10,000
m	99	9,900	999,799	475	49,296	4,992,271	1,204	124,684	12,496,704
$m_{\max}(G)$	4,950	499,500	49,995,000	4,950	499,500	49,995,000	4,950	499,500	49,995,000
$d(G)$	0.020	0.020	0.020	0.096	0.099	0.099	0.243	0.250	0.250
$\deg_{\min}(G)$	1	10	101	3	52	527	14	146	1,464
$\deg_{\max}(G)$	14	156	1,651	42	391	3,706	64	621	6,120
$\overline{\deg}(G)$	1.98	19.80	199.96	9.50	98.59	998.45	24.08	249.36	2,499.34
nodes.csv	291	3,891	48,891	291	3,891	48,891	291	3,891	48,891
edges.csv	634	82,465	10,330,489	3,119	420,815	52,603,962	8,209	1,096,976	134,950,863

intermediate), and dense graphs. As shown in Table 4.1, the column groups *2% Density*, *10% Density*, and *25% Density* represent the three categories we used based on the density of the graph. The columns G_1 through G_9 refer to the individually generated graphs, where G_1 to G_3 fall under 2% density, G_4 to G_6 fall under 10% density, and G_7 to G_9 fall under 25% density. The rows n and m provide the number of nodes and edges in the graphs, respectively; $m_{\max}(G)$ is the maximum number of edges that could be present in this graph; $d(G)$ provides the actual density of the graph to three decimal points (this can vary from the exact target density of the category because of randomness in the generation algorithm); $\deg_{\min}(G)$ is the minimum node degree in the graph; $\deg_{\max}(G)$ is the maximum node degree in the graph; and $\overline{\deg}(G)$ provides the average of all the nodes' degree in the graph. The rows *nodes.csv* and *edges.csv* specify the size of the files recording the graphs, in bytes. One common characteristic among the nine graphs is that none of them is a complete graph, which closely represents real-world scale-free graphs exhibiting a "long tail" (strong left-skew in the degree histogram).

Graphs can be represented and stored either as an adjacency list or an adjacency matrix. An

Listing 4.1: Script to generate graphs.

```
import networkx as nx
n = 100  # Number of nodes
m = 4  # Number of initial links
seed = 100
G = nx.barabasi_albert_graph(n, m, seed)
print(nx.info(G))
```



(a) sparse graph **(b)** semi-dense graph **(c)** dense graph

Figure 4.1: Examples of graphs with varying density.

adjacency list is a list where each list element represents a node; each element itself contains a list of the indices of the target node for each out edge. An adjacency matrix is a two-dimensional array where rows represent the source node and columns represent the target node. The cell at row i and column j contains the value 1 if the edge (v_i, v_j) is present in the graph; otherwise it is 0. In our study, we settled on using adjacency lists to represent our model data graphs, because:

- They are a compact option with a space complexity of $O(n)$ when compared to adjacency matrices which have a space complexity of $O(n^2)$ [Kleinberg, 2014].
- The savings in space increases dramatically if the dataset is sparse.
- Some database technologies have a limit on the number of columns that can exist in a table or relation, e.g., 1,024 columns per table for Microsoft SQL Server[1] and 4096 for MySQL.[2]
- The adjacency matrix permits random access (time complexity in $O(1)$), whereas the adjacency list requires a linear search (time complexity in $O(n)$) [Lafore, 2003].

4.3 Setup for evaluation

In this section, we explain in detail the setup for our evaluation, including the use cases analyzed and the database technologies used.

[1] `https://docs.microsoft.com/en-us/sql/sql-server/maximum-capacity-specifications-for-sql-server` [2021/06/29]
[2] `https://dev.mysql.com/doc/refman/8.0/en/column-count-limit.html` [2021/06/29]

4.3.1 Graph creation

In our experiment, we simulate the creation of the models for a SDAT by preparing a scale-free graph using the Barabási–Albert model. The creation of the benchmark model datasets and the underlying graphs is done by using the *networkx* library in Python. Listing 4.1 shows a code snippet which generates a basic scale-free graph for 100 nodes and ~400 edges, where `n` is the number of nodes, `m` is the number of initial links (i.e., the number of edges to attach from a new node to existing nodes), and `seed` is an indicator of random number generation state (we used the same value for the sake of reproducibility). Line 5 is where the function `barabasi_albert_graph()` is called which returns a graph `G`. Once all the graphs were generated they were then stored in two files: *nodes.csv* and *edges.csv* (not shown).

4.3.2 Database technologies used in the experiment

In our experiment, we supported semi-structured databases by using the *networkx* library[3] in Python 3.6.2 and comma-separated value (CSV) files. For relational database technology, we used MySQL Workbench (Version 8.0.22, Community Edition) and pgAdmin4 (v5.3) as a development platform for PostgreSQL which is a well known advanced object relational database technology that supports CASCADE. For non-relational database technology, we used Neo4j (v4.2) which is a graph database platform. The rest of this section describes the setup of the database technologies used.

4.3.2.1 Python-CSV

The *networkx* library has been used to create artificial datasets and the *csv* library is used to maintain the datasets with the changes implemented. We iterated a set of statements to implement the use-case and restored the dataset again back to the initial state within the loop. The execution time was captured for only those statements which were responsible to implement the use-case.

[3]Networkx. `https://networkx.github.io` [2021/06/29]

4.3.2.2 MySQL Workbench

To import our database, we used the `load data infile` method which reads from text files at a very fast speed. For the dataset containing nodes, we set the node id as the primary key to avoid duplicate entries of the same node again. The datasets were stored locally on the system and "–secure–file–**private**" option was disabled during the experiment to provide access to documents from the whole file system. MySQL workbench uses SQL language to define and manipulate data.

4.3.2.3 Neo4j

For small datasets, the direct import function available on the user interface of Neo4j can be used to import data into the database. For medium sized datasets, up to import to the database. For huge datasets, batch import command is used. Since our datasets fall in the range of medium-sized datasets, we used the load csv method to import the artificial datasets into the system. We chose to commit periodically after every 10,000 entries in order to avoid memory overflow. For optimization, we increased the page cache size and kept the maximum memory heap size to fifty percent of the total RAM minus the heap size. We constructed the dependency graph in a similar manner as the social media friendship graph, since both graphs exhibit long tail behavior. We utilized the node name as an index in the database to aid fast search. We also opted for "merge" rather than "create" to avoid redundancy in the node list. Neo4j uses the Cypher query language to define and manipulate graph and data.

4.3.2.4 pgAdmin 4

We set the node id as the primary key and the connection's source and target ids as foreign keys. We also made use of the update cascade and delete cascade methods in the edges table in order to automate a few processes such as renaming a node in the node; subsequently, the database would alter the entries in the edges database without manual intervention. This has also helped us to prevent adding new connections for the nodes that did not exist. To import the dataset to the database, we used the copy method.

4.3.3 Defining schema for MySQL and PostgreSQL

Listing 4.2 shows the setup required for the relational database technologies before we store the data in it. The command create table was used to create the table nodes, where id is the primary key. Similarly, we created to and from columns for the table edges, referencing the primary key of the nodes table. Lastly, we used create index to add indices on the to and from columns of table edges.

Listing 4.2: Defining schema and creating tables in PostgreSQL and MySQL.

```
create table nodes(
id INTEGER PRIMARY KEY
);

CREATE TABLE edges(
a INTEGER NOT NULL references nodes(id)
ON update cascade on delete cascade,
b INTEGER NOT NULL references nodes(id)
ON update cascade on delete cascade,
PRIMARY KEY(a,b)
);

create index a_idx on edges(a);
create index b_idx on edges(b);
```

4.3.4 Use cases realized via database technologies

Table 4.2 gives an overview of all the use cases implemented in this study. The *Operations* column describes the type of database operation being analyzed; these can be considered to be the standard "CRUD" operations which are further divided into sub-operations as shown in column *UC#* where

Table 4.2: Overview of use cases.

Operations	UC#	Description
Create a graph	UC I-1	Create or store a graph
Retrieve a graph	UC I-2	Read/access a graph
Update a graph	UC I-3	Create a node with no edges
	UC I-4	Create an edge between existing nodes
	UC I-5	Rename a node
	UC I-6	Change source and target nodes of an edge
Delete a graph	UC I-7	Delete a node and its corresponding edges
	UC I-8	Delete a specific edge

the first two operations are single use-case each, the third operation is comprised of four use cases, and the last operation consists of two use cases. There is a brief description in the *Description* column. These use cases are described in greater detail, below.

UC I-1: *Create or store a graph*. We used SQL to store the graphs in MySQL and PostgreSQL, and used Cypher to store them in Neo4j (they were already in the appropriate format for Python-CSV, so no explicit store operation was needed). We implemented all the use cases using Python to represent semi structured database creation and manipulation. For relational database technology,

Listing 4.3: Creating tables to store the graphs in PostgreSQL.

```
CREATE TABLE nodes (
    id INTEGER PRIMARY KEY,
    name VARCHAR(10) NOT NULL
);

CREATE TABLE edges (
    a INTEGER NOT NULL REFERENCES nodes(id)
        ON UPDATE CASCADE ON DELETE CASCADE,
    b INTEGER NOT NULL REFERENCES nodes(id)
        ON UPDATE CASCADE ON DELETE CASCADE,
    PRIMARY KEY (a, b)
);

CREATE INDEX a_idx ON edges (a);
CREATE INDEX b_idx ON edges (b);
```

Listing 4.4: UC I-1: Loading graphs into PostgreSQL.

```
COPY edges (a, b)
FROM '/Applications/friendship.csv' DELIMITER ',' CSV header;
```

Listing 4.5: UC I-1: Loading graphs into MySQL.

```
load data infile "/Applications/edges_100.csv"
into table edges fields terminated by ',' lines
terminated by '\n' IGNORE 1 LINES;
```

Listing 4.6: UC I-1: Loading graphs into Neo4j.

```
LOAD CSV WITH HEADERS FROM 'file:///people.csv' AS line WITH line
CREATE (:Person {id:line.id, name:line.name})
CREATE INDEX ON :Person(name);
USING PERIODIC COMMIT
```

Listing 4.7: UC I-2: Reading the graph data from Python-CSV.

```
with open('nodes.csv', 'r', newline='') as csvfile:
    node_data = csv.reader(csvfile, delimiter='\n', quotechar='|', quoting=csv.QUOTE_MINIMAL)
    nodes=list(node_data)
with open('edges.csv', 'r', newline='') as csvfile:
    edge_data = csv.reader(csvfile, delimiter=',', quotechar='|', quoting=csv.QUOTE_MINIMAL)
    edges=edge_data
```

Listing 4.8: UC I-2: Reading the graph data from Neo4j.

```
Match (n)-[r]->(m)
```

we first defined the schema, e.g., nodes and edges. Listing 4.3 contains the SQL query to define the schema of the database in PostgreSQL. As shown in Listing 4.4, once the relations were defined, we loaded our graphs into them, via `COPY` in PostgreSQL. For MySQL, Listing 4.5 shows how we used `load data infile` to store the graphs. Similarly for Neo4j, Listing 4.6 shows the loading of graphs using `CREATE` to create the nodes. Unlike with relational databases, in graph databases like Neo4j, the schema of a graph (or relation) does not need to be explicitly defined prior to storing the data. The relationships (edges) are created along the way while loading data from *.csv* files.

UC I-2: *Read/access a graph.* Once the database was created, we retrieved the entries to simulate the process of a data request from an SDAT to perform a user-requested analysis. For Python-CSV, we loaded the dataset from a CSV file to a data structure in memory as shown in Listing 4.7. For MySQL and Neo4j, all the rows of the "nodes" and "edges" tables were loaded into memory. Listing 4.8 shows how the nodes and relationships were typically loaded in Neo4j by only using the `Match` clause (the data loaded in the memory was not returned/displayed on the Neo4j browser, i.e., the "`return`" clause was not used).

UC I-3–UC I-6: Use cases UC I-3–UC I-6 simulate the modification of the model graph and its elements: adding a new node (UC I-3) as shown in Listing 4.9 for MySQL/PostgreSQL and in Listing 4.10 for Neo4j; adding a new edge (UC I-4), as shown in Listing 4.11 for MySQL/ PostgreSQL and in Listing 4.12 for Neo4j; renaming a node (UC I-5), as shown in Listing 4.13 for MySQL/PostgreSQL and in Listing 4.14 for Neo4j; and modifying an edge (UC I-6), as shown in

Listing 4.9: UC I-3: Creating a node via MySQL and PostgreSQL.

```
INSERT INTO nodes (id, name)
VALUES (1, '1');
```

Listing 4.10: UC I-3: Creating a node via Neo4j.

```
MERGE (:GRAPH { id: '23',name: '23' })
```

Listing 4.11: UC I-4: Creating a node via MySQL and PostgreSQL.

```
INSERT INTO edges (a,b)
VALUES (2, 7);
```

Listing 4.12: UC I-4: Creating an edge via Neo4j.

```
MATCH (to:GRAPH {name: '23'})
MATCH (from:GRAPH {name: '14'})
MERGE (to)-[:connects]->(from)
```

Listing 4.13: UC I-5: Renaming a node via MySQL and PostgreSQL.

```
UPDATE nodes SET id=6060  WHERE id=2;
UPDATE edges SET source=6060  WHERE source=2
UPDATE edges SET target=6060  WHERE target=2
```

Listing 4.14: UC I-5: Renaming a node via Neo4j.

```
MATCH (n:GRAPH {id: "1"}) SET n.name="new_name"
```

Listing 4.15: UC I-6: Changing the source and target of an edge via MySQL and PostgreSQL.

```
DELETE FROM edges
WHERE source = 2 and target = 3;
INSERT INTO edges (source, target)
VALUES (2, 42);
```

Listing 4.16: UC I-6: Changing the source and target of an edge via Neo4j.

```
MATCH (s:GRAPH { name: '13' })-[r:connects]->(t:GRAPH{name: '19'})
DELETE r
MATCH (source:GRAPH {name: '23'})
MATCH (target:GRAPH {name: '14'})
MERGE (source)-[:connects]->(target)
```

Listing 4.17: UC I-7: Deleting a node and its corresponding edges via MySQL and PostgreSQL.

```
MATCH (n:GRAPH) where n.name='3'
OPTIONAL MATCH (n)−[r]−()
DELETE n,r
```

Listing 4.18: UC I-7: Deleting a node and its corresponding edges via Neo4j.

```
MATCH (n:GRAPH) where n.name='3'
OPTIONAL MATCH (n)−[r]−()
DELETE n,r
```

Listing 4.19: UC I-8: Deleting a node and its corresponding edges via MySQL and PostgreSQL.

```
DELETE FROM edges
WHERE source = 2 and target = 3;
```

Listing 4.20: UC I-8: Deleting a node and its corresponding edges via Neo4j.

```
MATCH (n:GRAPH {name: '13'})−[r:connects]−>(n:GRAPH{name: '19'})
DELETE r
```

Listing 4.15 for MySQL/PostgreSQL and in Listing 4.16 for Neo4j.

UC I-7–UC I-8: The remaining use cases simulate the deletion process of the model graph and its elements: deleting a node (UC I-7), as shown in Figure 4.17 for MySQL/PostgreSQL and in Figure 4.18 for Neo4j; and deleting an edge (UC I-8), as shown in Figure 4.19 for MySQL/PostgeSQL and in Figure 4.20 for Neo4j.

4.3.5 Time measurement

The ModCP prototype tool is a Windows Forms application and thus, it uses a single-threaded apartment model.[4] We specified a single processor to run the threads of this process to improve the performance by reducing the number of times the processor cache is reloaded.[5] As shown in Listing 4.21, ProcessorAffinity was used to associate the threads of the process to a single processor. Then, we set the overall priority of the above associated process to high by using ProcessPriorityClass.High.

[4]COM threading model. `https://docs.microsoft.com/en-us/previous-versions/dotnet/netframework-3.0/ms182351(v=vs.80)?redirectedfrom=MSDN` [2021/08/24]

[5]Processor affinity `https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics.process.processoraffinity?view=net-5.0` [2021/08/24]

Listing 4.21: Setting the processor priority and affinity for running the experiment.

```
var testProcess = Process.GetCurrentProcess();
testProcess.ProcessorAffinity = (System.IntPtr)1;
testProcess.PriorityClass = ProcessPriorityClass.High;
```

Each of the use cases were implemented on the candidate database technologies ten times, and the time taken to process the query were recorded from the user interface. The average of query processing time to implement a use-case taken by the each database was recorded to evaluate the database technologies. Similarly, the space measurement was noted from the user interface of the database technology.

4.3.6 System information

The system used for performing the experiment runs Microsoft Windows 10 Enterprise. It has Intel(R) Core(TM) i7-7700 CPU @ 3.60 GHz, 3600 MHz, 4 cores, 8 logical processors, and 8 GB of RAM. During the experiment, no user programs other than the test programs were running. A basic internet connection was on but not used during the experiment.

4.4 Results and analysis

We describe and analyze the results from our objective and subjective evaluation for the databases. Results are provided in terms of the measures used for each of the evaluations.

4.4.1 Objective evaluation

In this section we discuss the results of a study conducted to evaluate the performance of the selected database technologies. The complete results of this study are shown in Table 4.3. We examine the results for individual use cases in subsequent subsections.

In all the plots we present for this comparative study:

Table 4.3: Time taken to process the use cases, in milliseconds.

Dens.	Gr.	Technology	UC I-1	UC I-2	UC I-3	UC I-4	UC I-5	UC I-6	UC I-7	UC I-8
2%	G_1	Python-CSV	**0**	**1**	**1**	**1**	**2**	**2**	**3**	**2**
		MySQL	5	2	153	156	155	155	156	154
		Neo4j	665	18	3	3	6	7	11	**2**
		PostgreSQL	187	30	61	66	63	63	66	65
	G_2	Python-CSV	**0**	**11**	**3**	8	41	13	39	12
		MySQL	188	13	151	154	152	158	155	155
		Neo4j	6,852	223	**3**	**3**	**11**	**6**	**13**	**3**
		PostgreSQL	236	157	63	67	62	63	64	66
	G_3	Python-CSV	**0**	1,923	325	19	2,459	1,694	2,281	1,659
		MySQL	14,763	965	152	153	152	156	155	157
		Neo4j	47,970	1,753	**4**	**3**	**14**	**7**	**12**	**3**
		PostgreSQL	12,989	**1,586**	63	66	65	67	65	66
10%	G_4	Python-CSV	**0**	**2**	**2**	4	**3**	**3**	**4**	3
		MySQL	27	**2**	157	153	152	156	155	156
		Neo4j	2,172	157	3	**3**	8	7	13	**2**
		PostgreSQL	214	138	62	65	62	65	64	65
	G_5	Python-CSV	**0**	86	**3**	153	103	68	70	72
		MySQL	694	**57**	155	154	158	157	157	159
		Neo4j	8,936	480	4	**4**	**9**	**7**	**13**	**3**
		PostgreSQL	519	387	61	66	64	67	66	66
	G_6	Python-CSV	**0**	9,327	358	3,093	7,231	7,295	8,640	7,411
		MySQL	74,971	4,294	158	153	152	157	154	159
		Neo4j	357,118	2,935	**3**	**4**	**13**	**6**	**15**	**4**
		PostgreSQL	64,322	**1,822**	63	66	66	64	63	66
25%	G_7	Python-CSV	**0**	**4**	**2**	**3**	**3**	**4**	**3**	**3**
		MySQL	481	11	153	155	157	156	156	155
		Neo4j	2,328	396	3	4	8	7	12	**3**
		PostgreSQL	498	284	60	65	63	65	64	65
	G_8	Python-CSV	**0**	163	7	83	281	170	174	89
		MySQL	1,572	**52**	153	155	157	156	155	157
		Neo4j	25,739	1,067	**3**	**4**	**12**	**8**	**14**	**3**
		PostgreSQL	1,325	819	59	66	64	65	66	64
	G_9	Python-CSV	**0**	14,216	363	8,954	26,227	22,683	19,593	20,279
		MySQL	130,136	21,980	154	155	161	157	155	158
		Neo4j	1,262,200	8,139	**4**	**4**	**15**	**7**	**13**	**4**
		PostgreSQL	121,820	**5,128**	64	65	65	66	64	67

1. we sort the data according to the number of edges in the graphs, resulting in the following sequence: G_1, G_4, G_7, G_2, G_5, G_8, G_3, G_6, and G_9;

2. we plot edge count on the x-axis and time taken in milliseconds on the y-axis;

3. we plot the data on log–log scales because the core results grow rapidly, otherwise obscuring their trends; and,

4. we prioritize the number of edges over the number of nodes to compare the database technologies, as number of edges tends towards being quadratic in the number of nodes, hence dominating.

Furthermore, we attempt to fit a linear model $\log y = p \log x + k$ (base e) to the log–log data, sorted by edge count, for each technology/use case combination. Because the lower ends of this data involve numbers of edges that are linear in the number of nodes, we consider only the uppermost six data points for each in fitting the linear model. We note that such a procedure has potential statistical imprecision, but suffices for the trend comparison between technologies in which we are interested. We report values for the *coefficient of determination* $R^2 \in [0,1]$ (the proportion of the variance in the dependent variable that is predictable from the independent variable), but we acknowledge that this gives only some information about the goodness of fit of the model: low values can occur for well-fitting models (e.g., when the fitted line is nearly parallel with the x-axis) and high values can occur even when the fit is not obviously good. We also consider the visual fit in cases where our analyses depend on the evaluation of the model.

4.4.1.1 Examination of the use cases

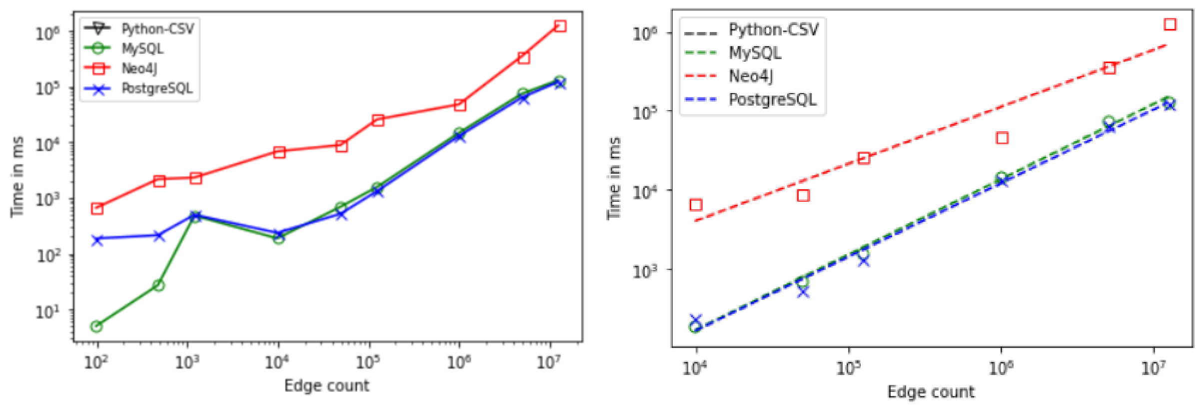
UC I-1: *Create or store a graph.* Figure 4.2a shows the time taken by the individual database technologies to store the artificially generated graph; Python-CSV does not require any time to store the model as the dataset is already in CSV format, so its data does not appear on the plot (the logarithm of 0 is undefined). Figure 4.2b shows the results of the linear regressions on the larger graphs; Table 4.4 shows the observed computation times for each graph, adjacent to the predicted computation times from the fitted linear regression model.

We see that MySQL and PostgreSQL are roughly collinear (their linear regressions place their slopes at 0.95 and 0.93 with constant -3.66 and -3.50, respectively). Neo4j clearly has higher overhead (linear regression constant is 1.65) but its computation time appears to grow slightly more slowly than for the SQL variants (linear regression slope is only 0.72): extrapolating the fitted models, we would expect the time for Neo4j and PostgreSQL to be equal when the edge count be in the vicinity of 4.5×10^{10}. Obviously, this assumes that the growth characteristics can be extrapolated in this manner; but, aside from the dubiousness of this extrapolation (the fitted model for Neo4j has an R^2 of only 0.73 and visually is not a great fit), this intersection point would only occur for truly enormous graphs, far beyond any we have encountered or that are likely in practice, even in extreme situations. Python-CSV obviously outperforms all other competition, but this is a local anomaly as we will see examining the results for the other use cases.

UC I-2: *Read/access a graph.* Figure 4.3a shows the time taken to retrieve the graphs from the database into memory. Both MySQL and Python-CSV were inexpensive to retrieve the model for graphs with relatively few edges when compared to PostgreSQL and Neo4j; however, the costs for all the technologies remained low ($\leq$1,067 ms for Neo4j) even in the worst case.

From Figure 4.3b we can see that the steeper slopes of MySQL and Python-CSV lead to a crossover point at around 10^6 edges when these two technologies become more expensive to use than Neo4j and PostgreSQL. Table 4.5 shows the observed computation times for each graph, adjacent to the predicted computation times from the fitted linear regression model. We found that MySQL and Python-CSV have approximately the same cost (their slopes are 1.02 and 1.03 with constant -6.78 and -7.34, respectively). Similarly, Neo4j and PostgreSQL exhibit similar growth patterns (their slopes are 0.45 and 0.43 with constant 1.31 and 1.29, respectively). Based on these observations, the best database technology for smaller graphs is either MySQL or Python-CSV, with PostgreSQL or Neo4j being preferable for larger graphs (above 10^6 edges), for UC I-2.

UC I-3: *Update a graph; Create a node with no edges*. Figure 4.4a shows the performance comparison of the candidate database technologies to create a new node in an existing graph.

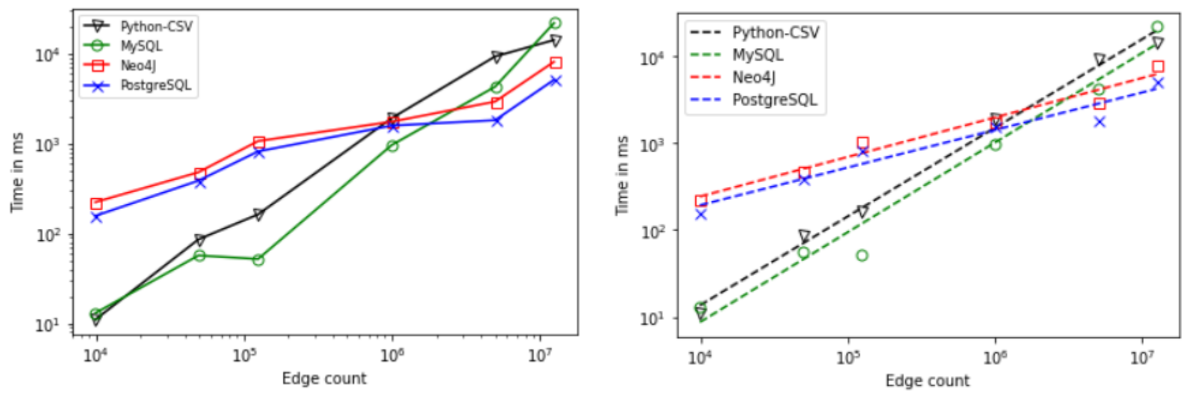


(a) Raw data. **(b)** Linear regression results.

Figure 4.2: Comparison of edge count versus computation time for UC I-1.

Table 4.4: Computation time to perform UC I-1 per graph relative to data storage technology. Observed values and values obtained from the fitted model $\log(time) = p\log(edge\text{-}count) + k$ via linear regression on the log–log data and coefficient of determination R^2.

	G_2	G_5	G_8	G_3	G_6	G_9	p	k	R^2
Python-CSV									
observed	0	0	0	0	0	0			
modelled	—	—	—	—	—	—	—	—	—
MySQL									
observed	188	694	1,572	14,763	74,971	130,136			
modelled	165.75	765.31	1,853.04	13,473.28	62,376.07	149,546.57	0.95	-3.66	0.96
Neo4j									
observed	6,852	8,936	25,739	47,970	357,118	1,262,200			
modelled	3,978.89	12,677.47	24,771.13	111,324.37	355,413.59	689,282.84	0.72	1.65	0.73
PostgreSQL									
observed	236	519	1,325	12,989	64,322	121,820			
modelled	161.99	724.25	1,721.28	12,003.23	53,805.76	126,645.80	0.93	-3.50	0.99



(a) Raw data. **(b)** Linear regression results.

Figure 4.3: Comparison of edge count versus computation time for UC I-2.

Table 4.5: Computation time to perform UC I-2 per graph relative to data storage technology. Observed values and values obtained from the fitted model $\log(time) = p\log(edge\text{-}count) + k$ via linear regression on the log–log data and coefficient of determination R^2.

	G_2	G_5	G_8	G_3	G_6	G_9	p	k	R^2
Python-CSV									
observed	11	86	163	1,923	9,327	14,216			
modelled	13.56	69.77	179.82	1,504.32	7,761.34	19,794.44	1.02	-6.78	0.82
MySQL									
observed	13	57	52	965	4,294	21,980			
modelled	8.68	45.51	118.59	1,016.75	5,346.36	13,783.86	1.03	-7.34	0.82
Neo4j									
observed	223	480	1,067	1,753	2,935	8,139			
modelled	243.25	504.37	768.8	1,979.31	4,109.24	6,234.23	0.45	1.31	0.88
PostgreSQL									
observed	157	387	819	1,586	1,822	5,128			
modelled	193.26	386.74	577.52	1,419.90	2,844.79	4,229.15	0.43	1.29	0.88

Figure 4.4b shows the results of the linear regressions on the larger graphs; Table 4.6 shows the observed computation times for each graph, adjacent to the predicted computation times from the fitted linear regression model.

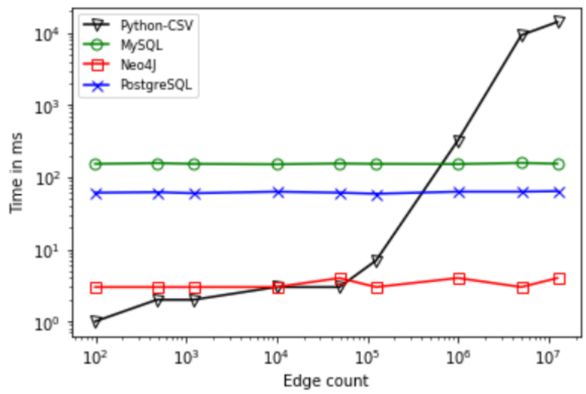
We see that Python-CSV and Neo4j have the lowest costs cost for G_2, but Python-CSV then scales poorly, leading to the highest cost for G_9 (Python-CSV shows a high linear regression slope of 1.38). We also found that MySQL, Neo4j, and PostgreSQL take essentially constant time to realize the use case (their linear regressions place their slopes at 0.01, 0.02, and 0.01, respectively).

Neo4j is the best option for UC I-3 as the base cost to create a node is the cheapest and remains constant for larger graphs. The slightly higher but constant cost for PostgreSQL makes a viable alternative as well, relative to UC I-3.

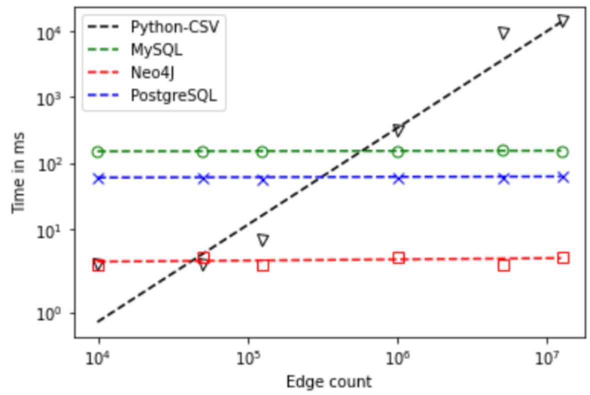
UC I-4: *Update a graph; Create an edge between existing nodes*. Figure 4.5a shows the time taken by the individual database technologies to create edges in the edge relation (for relational database technologies) as well as in the link database (for non relational database technologies). Figure 4.5b shows the results of the linear regressions on the larger graphs; Table 4.7 shows the observed computation times for each graph, adjacent to the predicted computation times from the fitted linear regression model.

We see that Python-CSV has near linear growth with respect to edge count (slope is 0.81) and so it is not competitive with the alternatives. MySQL and PostgreSQL take roughly the same time (slopes are 0.01 and 0.01, with constants 5.04 and 4.23, respectively). Neo4j also shows essentially constant performance (slope is 0.02) with lower fixed cost (constant is 1.02). Relative to UC I-4, MySQL, PostgreSQL, and Neo4j are all viable candidates.

UC I-5: *Update a graph; Rename a node*. Figure 4.6a shows the time taken by the individual database technologies to rename the nodes in the node and edge relation (for relational database technologies) as well as in the link database (for non-relational database technologies). Figure 4.6b shows the results of the linear regressions on the larger graphs; Table 4.8 shows the observed computation times for each graph, adjacent to the predicted computation times from the fitted linear regression model.



(a) Raw data.

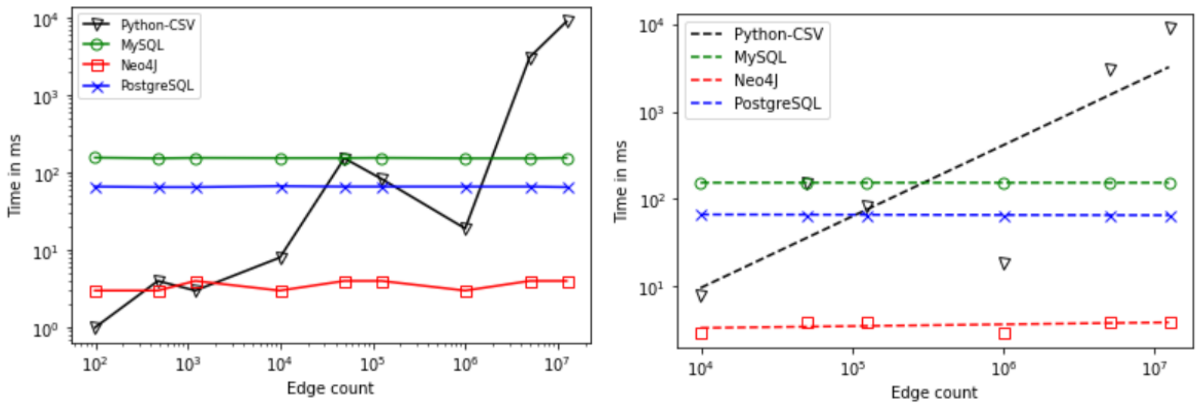


(b) Linear regression results.

Figure 4.4: Comparison of edge count versus computation time for UC I-3.

Table 4.6: Computation time to perform UC I-3 per graph relative to data storage technology. Observed values and values obtained from the fitted model $\log(time) = p\log(edge\text{-}count) + k$ via linear regression on the log–log data and coefficient of determination R^2.

	G_2	G_5	G_8	G_3	G_6	G_9	p	k	R^2
Python-CSV									
observed	3	3	7	325	9,327	14,216			
modelled	0.74	6.72	24.12	424.63	3,892.02	13,777.95	1.38	-12.98	0.85
MySQL									
observed	151	155	153	152	158	154			
modelled	152.07	152.83	153.27	154.26	155.03	155.48	0.01	5.00	0.29
Neo4j									
observed	3	4	3	4	3	4			
modelled	3.25	3.34	3.4	3.52	3.62	3.68	0.02	1.02	0.09
PostgreSQL									
observed	63	61	59	63	63	64			
modelled	60.98	61.48	61.78	62.44	62.96	63.25	0.01	4.06	0.23



(a) Raw data. **(b)** Linear regression results.

Figure 4.5: Comparison of edge count versus computation time for UC I-4.

Table 4.7: Computation time to perform UC I-4 per graph relative to data storage technology. Observed values and values obtained from the fitted model $\log(time) = p\log(edge\text{-}count) + k$ via linear regression on the log–log data and coefficient of determination R^2.

	G_2	G_5	G_8	G_3	G_6	G_9	p	k	R^2
Python-CSV									
observed	8	153	83	19	3,093	8,954			
modelled	9.65	35.66	75.91	413.45	1,531.3	3,232.36	0.81	-5.22	0.45
MySQL									
observed	154	154	155	153	153	155			
modelled	154.11	154.06	154.03	153.97	153.92	153.89	0.01	5.04	0.01
Neo4j									
observed	3	4	4	3	4	4			
modelled	3.37	3.48	3.55	3.70	3.83	3.90	0.02	1.02	0.14
PostgreSQL									
observed	67	66	66	66	66	65			
modelled	66.68	66.38	66.21	65.83	65.53	65.36	0.01	4.23	0.66

We see that Python-CSV involves the highest query processing times (slope is 0.91 with constant -4.95). In contrast, MySQL, Neo4j and PostgreSQL show essentially constant performance (slopes are 0.01, 0.05, and 0.01, respectively) although the fixed cost is somewhat higher for MySQL (constants are 5.02, 1.82, and 4.1, respectively).

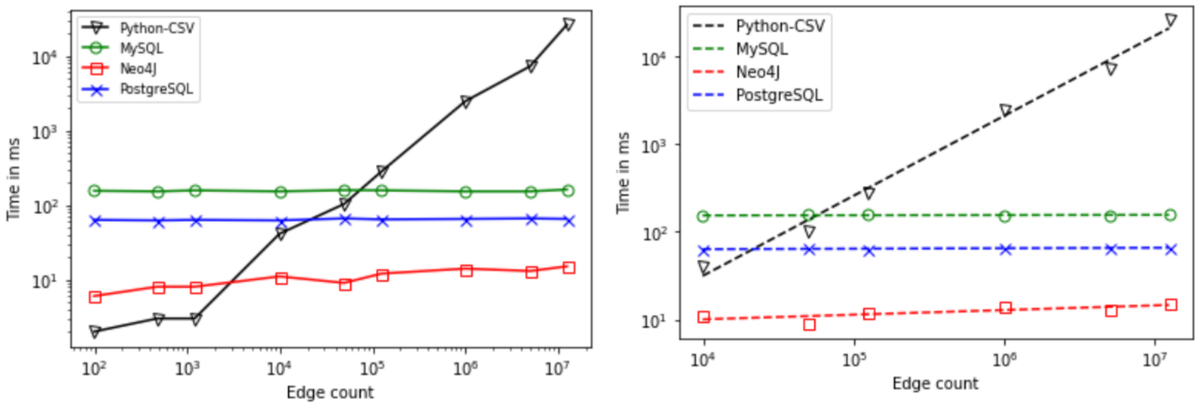
UC I-6: *Update a graph; Change source and target nodes of an edge*. Figure 4.7a shows the time taken by the individual database technologies to update a graph stored therein by modifying an existing edge. This involves changing the source and target node of an edge in the database. Figure 4.7b visualizes the results of the linear regressions; Table 4.9 shows the observed computation times for each graph, adjacent to the predicted computation times from the fitted linear regression model.

Python-CSV shows linear growth (slope is 1.04) making it unsuitable for larger graphs. We observe that both MySQL and PostgreSQL require constant processing time (slope is 0.01 for both). Neo4j has lower fixed cost than PostgreSQL (constants are 1.91 and 4.15, respectively).

UC I-7: *Delete a graph; Delete a node and its corresponding edges*. Figure 4.8a shows the time taken by the individual database technologies to delete a node. We performed the operation by first deleting the node and then deleting the edges related to it. Figure 4.8b visualizes the results of the linear regressions; Table 4.10 shows the observed computation times for each graph, adjacent to the predicted computation times from the fitted linear regression model.

The performance of Python-CSV is competitive only for the smaller graphs; its linear growth (slope is 0.94) excludes it from further consideration with respect to UC I-7. MySQL, Neo4j and PostgreSQL provide essentially constant performance (slopes are 0.01, 0.02 and 0.01, respectively). However, the fixed cost for Neo4j is better than for PostgreSQL or MySQL (constants are 5.06, 2.29, and 4.21, respectively).

UC I-8: *Delete a graph; Delete a specific edge*. Figure 4.9a shows the time taken by the individual database technologies to delete a specific edge. This involved changing the source and target node of an edge in the database. Figure 4.9b visualizes the results of the linear regres-

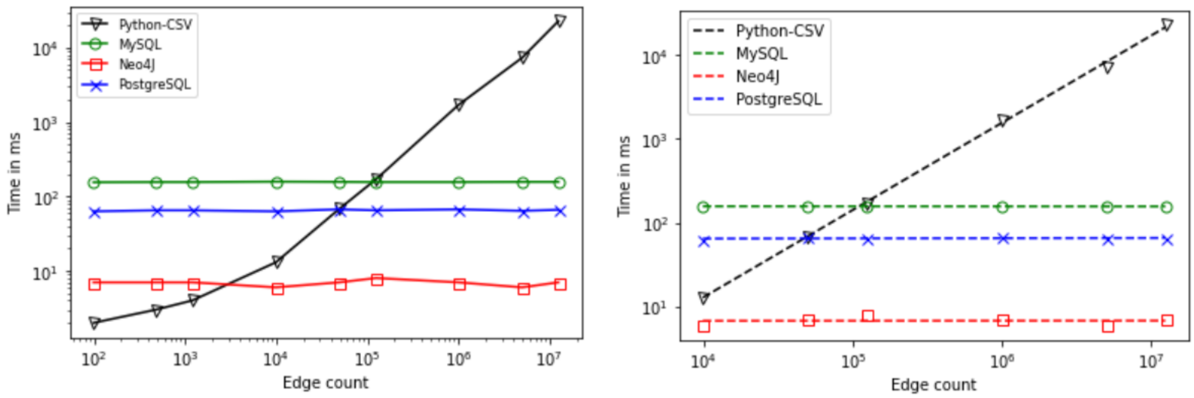


(a) Raw data. **(b)** Linear regression results.

Figure 4.6: Comparison of edge count versus computation time for UC I-5.

Table 4.8: Computation time to perform UC I-5 per graph relative to data storage technology. Observed values and values obtained from the fitted model $\log(time) = p\log(edge\text{-}count) + k$ via linear regression on the log–log data and coefficient of determination R^2.

	G_2	G_5	G_8	G_3	G_6	G_9	p	k	R^2
Python-CSV									
observed	41	103	281	2,459	7,231	26,227			
modelled	31.37	135.72	316.46	2,114.43	9,170.24	21,181.57	0.91	-4.95	0.94
MySQL									
observed	152	158	157	152	152	161			
modelled	154.10	154.62	154.92	155.60	156.12	156.42	0.01	5.02	0.06
Neo4j									
observed	11	9	12	14	13	15			
modelled	10.01	10.9	11.45	12.78	13.91	14.61	0.05	1.82	0.68
PostgreSQL									
observed	62	66	64	65	66	65			
modelled	63.43	63.96	64.27	64.96	65.51	65.82	0.01	4.1	0.37



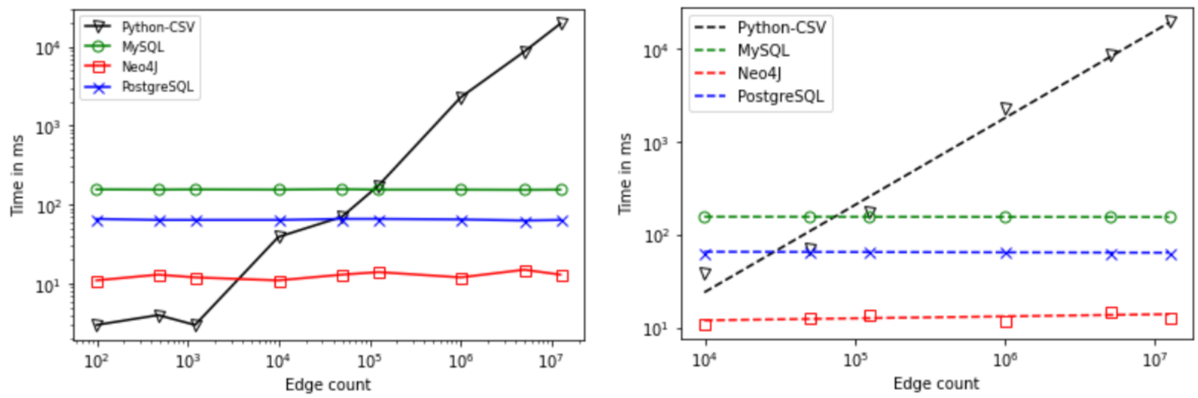
(a) Raw data.

(b) Linear regression results.

Figure 4.7: Comparison of edge count versus computation time for UC I-6.

Table 4.9: Computation time to perform UC I-6 per graph relative to data storage technology. Observed values and values obtained from the fitted model $\log(time) = p\log(edge\text{-}count) + k$ via linear regression on the log–log data and coefficient of determination R^2.

	G_2	G_5	G_8	G_3	G_6	G_9	p	k	R^2
Python-CSV									
observed	13	68	170	1,694	7,295	22,683			
modelled	12.83	67.96	178.14	1,547.55	8,220.48	21,316.97	1.04	-7	0.99
MySQL									
observed	158	157	156	156	157	157			
modelled	157.16	157.02	156.94	156.75	156.60	156.52	0.01	5.06	0.11
Neo4j									
observed	6	7	8	7	6	7			
modelled	6.78	6.79	6.79	6.80	6.81	6.81	0.01	1.91	0.01
PostgreSQL									
observed	63	67	65	67	64	66			
modelled	64.73	64.99	65.13	65.46	65.72	65.87	0.01	4.15	0.07



(a) Raw data.

(b) Linear regression results.

Figure 4.8: Comparison of edge count versus computation time for UC I-7.

Table 4.10: Computation time to perform UC I-7 per graph relative to data storage technology. Observed values and values obtained from the fitted model $\log(time) = p\log(edge\text{-}count) + k$ via linear regression on the log–log data and coefficient of determination R^2.

	G_2	G_5	G_8	G_3	G_6	G_9	p	k	R^2
Python-CSV									
observed	39	70	174	2,281	8,640	19,593			
modelled	23.91	107.45	256.14	1,798.28	8,103.14	19,129.71	0.94	-5.44	1.00
MySQL									
observed	155	157	155	155	154	155			
modelled	155.81	155.53	155.36	155.00	154.72	154.56	0.01	5.06	0.24
Neo4j									
observed	11	13	14	12	15	13			
modelled	11.96	12.37	12.62	13.19	13.65	13.92	0.02	2.29	0.27
PostgreSQL									
observed	64	66	66	65	63	64			
modelled	65.44	65.10	64.90	64.46	64.13	63.94	0.01	4.21	0.23

sions; Table 4.11 shows the observed computation times for each graph, adjacent to the predicted computation times from the fitted linear regression model.

Python-CSV displays near-linear growth in performance (slope is 1.05) and is competitive only for the smallest graphs. MySQL, Neo4j, and PostgreSQL display essentially constant performance (slopes are 0.01, 0.04, and 0.01, respectively), although the fixed cost of Neo4j is lower (constants are 5.03, 0.62 and 4.16, respectively).

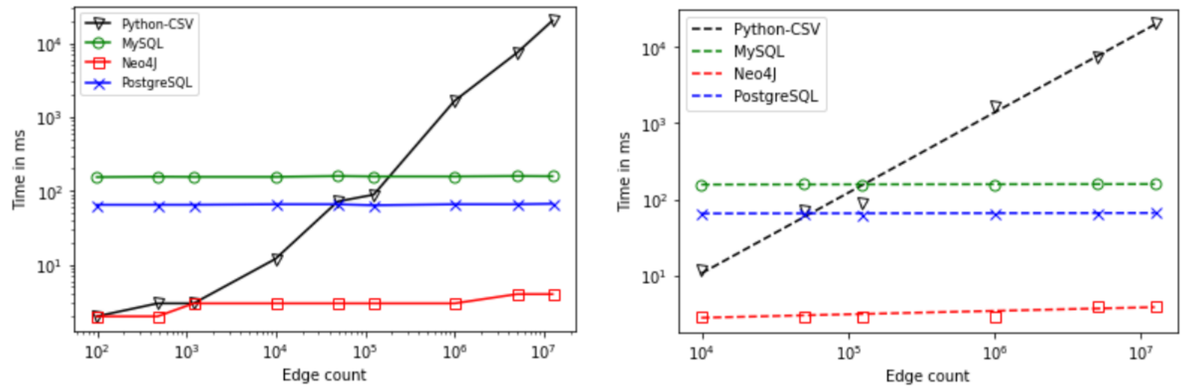
4.4.2 Subjective evaluation

In this section, we present the results of our subjective evaluations. Subjective measures are difficult to assess, but they are a crucial factor for a company to adopt and implement a database technology. We discuss in detail the information that we gathered in the rest of this section and also how the technologies under consideration perform for specific measures.

4.4.2.1 Maturity and level of support

Maturity is the amount of time a technology is offered by a vendor to a customer with a promise of an efficient and bug free environment. The longer a technology is in the market, the more feedback it will receive and the more it will be tested and maintained over time. This process refines the offered technology/software and can then be considered more reliable and stable to use as an implementation. When comparing the maturity of relational databases versus non-relational databases, the relational database will generally have the upper hand, having been in existence for longer.

The level of support goes side by side with maturity. So, the more mature a technology/software/system is, the higher the level of support is to be found. MySQL and PostgreSQL both use syntax query language (SQL) which is common amongst relational database technologies, thus the introduction to relational database technology is considered simple. Whereas non-relational databases do not have a common language to operate. Neo4j is a relatively new graph-based database technology that uses its novel Cypher query language. Cypher, being a non-standard and



(a) Raw data. **(b)** Linear regression results.

Figure 4.9: Comparison of edge count versus computation time for UC I-8.

Table 4.11: Computation time to perform UC I-8 per graph relative to data storage technology. Observed values and values obtained from the fitted model $\log(\mathit{time}) = p\log(\mathit{edge\text{-}count}) + k$ via linear regression on the log–log data and coefficient of determination R^2.

	G_2	G_5	G_8	G_3	G_6	G_9	p	k	R^2
Python-CSV									
observed	12	72	89	1,659	7,411	20,279			
modelled	10.67	57.84	153.66	1,375.60	7,478.88	19,652.67	1.05	-7.32	1
MySQL									
observed	155	159	157	157	159	158			
modelled	156.37	156.86	157.14	157.78	158.27	158.56	0.01	5.03	0.31
Neo4j									
observed	3	3	3	3	4	4			
modelled	2.80	3.01	3.14	3.44	3.70	3.85	0.04	0.62	0.73
PostgreSQL									
observed	66	66	64	66	66	67			
modelled	65.28	65.52	65.65	65.97	66.21	66.35	0.01	4.16	0.18

a new language, it has a learning phase involved for developers. However, the Cypher queries are shorter and quicker to write compared to SQL [Francis et al., 2018]. Neo4j is still in its infancy and has a limited level of support, mostly through its website.[6]

4.4.2.2 Security

Until recently, Neo4j had no built-in mechanism for security measures and multi-user management. Over time, Neo4j has implemented the following security mechanisms[7]:

1. Deploying Neo4j on only those servers which are safe.
2. Use of subnets and firewalls.
3. Securing the ports such that the port being used has no external access reducing potential piracy of the database.
4. Measures to safeguard data such as volume encryption and access control to data files and transactions.
5. For remote access to Neo4j database, the user is recommended to use secure http and SSL certificates which ensures security of in-transit data.

Apart from these mechanisms, Neo4j recommends that users follow a set of guidelines to ensure maximum security such as: checking correct file permissions; restricting access to bin, lib, and plugins directories to prevent execution of unauthorized extensions; and configuring the `load_csv` function properly to restrict unauthorized import calls using Neo4j authorization. Neo4j has provided multi-user support only recently despite many past claims by any past research papers and websites to the contrary.

MySQL, on the other hand, provides the following security measures[8]:

1. Plugins for authenticating the clients.

[6] Neo4j Support. `https://www.neo4j.org` [2021/06/29]
[7] Neo4j Security Mechanisms. `https://neo4j.com/docs/operations-manual/current/security/checklist/` [2021/06/29]
[8] MySQL Security Mechanisms. `https://dev.MySQL.com/doc/MySQL-security-excerpt/5.7/en/security-plugins.html` [2021/06/29]

2. A password-validation plugin to check the security level of user-chosen passwords and implementing password policies.

3. A key-ring plugin to deliver a secure database storage for sensitive and confidential information

MySQL features some stricter security measures which are only limited to the enterprise version. These include: (1) a firewall plugin which provides safety against attacks on the server including SQL injection or any attempt to misuse the application which lies out of their allowed and usual query characteristics, (2) data masking and de-identification by means of a plugin to conceal the sensitive information by altering the real values and replacing them with substitutes, and (3) an audit plugin to keep a track of connection logs, query activity, and warrant standard policy-based monitoring.

Security in PostgreSQL is achieved mostly by restricting access privileges and permissions at the user level. Files in the database can not be accessed by any account other than the superuser. Client user authentication can be managed through the `pg_hba.conf` file in `PG_DATA` which contains database user credentials including name and IP addresses.[9] On top of authentication control, connections from clients are only permitted via local UNIX socket when the back-end is initialized using the "–i" option, which restricts non-local clients to access database. By default a user who is not the creator of the database does not have a write access. Users can be grouped to ease the process managing privileges. PostgreSQL currently does not have any password management security measure. Users are always advised to opt for a strong password and to change it periodically to thwart password hacks. External authentication tools like Lightweight Directory Access Protocol (LDAP) can be used to mandate a strong password, password expiration, and account locks on multiple incorrect password errors.

[9] PostgreSQL Security Mechanisms. https://www.PostgreSQL.org/docs/7.0/security.html [2021/06/29]

4.4.2.3 Flexibility

The model data is encoded in a schema to store it in an efficient manner. The schema of a database enables the user to structure the data and query in an organized fashion. Schema in relational database tends to be rigid and difficult to manage/change once the modelled data has already been structured and stored. However, in graph database Neo4j, the schema is defined at the same time as the data is stored. This allows the alteration of the whole database structure with minimum cost, and greater flexibility to the user in case of change in the model data structure in the future.

4.5 Discussion

In this section we discuss remaining issues relative to this study.

4.5.1 Space

We did not include the space taken by the graph in the database technology as an evaluation measure because there was no significant difference between them, as illustrated in Table 4.12 when measured for G_9. For Python-CSV, the size of the dataset is identical to the size of the database. For PostgreSQL and MySQL, the database size was almost identical. For Neo4j, the size of the database was slightly higher than for MySQL and PostgreSQL, but it was not significant enough to be used as an indicator of definite better performance.

Table 4.12: Space requirements (in bytes) for database technologies, relative to G_9.

Technology	Space
MySQL	514,234,210
PostgreSQL	534,520,112
Neo4j	583,381,354

4.5.2 Deleting the graph

As shown in Listing 4.22, `Truncate` was used for clearing the graph data while maintaining the relation schema for future storage of the graphs. Listing 4.23 shows the process of deleting all the nodes and relationships from the database while maintaining the labels used such as “GRAPH”.

4.5.3 Other datastore technologies

When we were selecting the candidate datastore technologies for our study, we focused on picking one technology from relational databases and one from non-relational databases. As for relational databases, we picked MySQL as being the most popular free RDBMS. We also included PostgreSQL, an object-based RDBMS, in our study as it is widely known as “the world’s most advanced RDBMS database,” and on the fact that not many comparative studies have been conducted to compare it with MySQL. The existing studies focus on the difference between the features they offer instead of on their actual performance differences [Poljak et al., 2017]. We picked up Neo4j as it is a graph database and made specifically to deal with graphs.

While careful consideration was undertaken when designing the study plan, we recognize that other datastore options could have been valid. For non-relational databases, MongoDB, a popular document-based NoSQL database, could also have been a better alternative based on its ability to support huge volumes of both data and traffic. For relational databases, Oracle could have been a good alternative choice. In the end, it was not feasible to try all alternatives and so a selection was needed.

Listing 4.22: Deleting the graph from the MySQL and PostgreSQL database.

```sql
TRUNCATE TABLE edges;
```

Listing 4.23: Deleting the graph from the Neo4j database.

```
MATCH (n)
DETACH DELETE n
```

4.5.4 Threats to validity

In this section we discuss the threats to validity of our work.

4.5.4.1 Internal Validity

One of the most common threats in empirical research exists against internal validity while also being one of the more difficult ones to counter. Changes in the variable under observation could very well be caused by additional variables or variations in such variables, which may be related to the manipulated variable but not explicitly modelled [Wright et al., 2010].

Selection Bias In this study, we had to make choice while selecting an artificial graph generation model. We selected the Barabási–Albert model as a representation of random scale-free graphs. Other variations of growth network model can be found in Buckley and Osthus [2004] which presents a directed preferential attachment model. Dorogovtsev et al. [2000] and Drinea et al. [2001] introduced a variation on the Barabási–Albert model. There also exist other models such as the "copying model" by Kumar et al. [2000], Erdős and Rényi [1960] present a random graph model which is much smaller than many real-world applications. We concluded that due to having been extensively studied from heuristic and experimental points of view made Barabási–Albert model a reliable choice to generate the datasets.

4.5.4.2 External Validity

We define external validity as referring to the generalization of the results, i.e., whether it would be considered "safe" to apply the results of a software study to all software of that type [Calder et al., 1982]. Threats to external validity are conditions that limit the ability to generalize the study results and we explain these threats in this section.

Selection of Subjects As discussed in Section 3, there are four major kinds of non-relational database. However, we did not evaluate all four kinds in our work. Our goal was to evaluate multiple database technologies to find if any of the choices would be a good fit to implement offline storage mechanism in SDATs. We evaluated relational and non-relational databases both objectively and subjectively and the study plan was inline with other existing work and therefore we can be confident in applying the results of our study to other SDATs without loss of generality [Petri, 2005; Batra and Tyagi, 2012; Rautmare and Bhalerao, 2016].

Settings Threats to external validity may also arise from the environment in which the experiments are conducted. The extent to which the results of an experiment can be generalized from the set of environmental conditions created by the researcher to other environmental conditions can greatly impact the generalizability of the results. The details of the windows machine used to conduct this study has been provided in Section 4.3.6. We evaluated this study, as well as the latter studies, on the same Windows machine.

4.5.4.3 Conclusion Validity

Every empirical study establishes relationships between the treatment, represented by the independent variables, and the outcomes, represented by the dependent variables [Zhou et al., 2016]. Conclusion validity refers to the belief in the ability to derive conclusions from the relationships between the independent variables and the dependent variable.

Reliability of Measures Reliability depends on a variety of factors, including, but not limited to, poor question wording, bad instrumentation, and subjective measures. Threats to conclusion validity may be caused by these factors. In our work, we used objective as well as subjective measures to evaluate the storage mechanisms. However, subjective measures may be considered to be unreliable as it requires the judgment of the researcher Ampatzoglou et al. [2019].

The objective measures consist of time and space taken to process a use case. Our results for this study uses average of all the runs; the actual processing time may be lower or higher than the average run time. The subjective measures used to evaluate the storage mechanisms may not accurately represent the measures being used universally in industrial practice[Herbsleb and Mockus, 2003]. However, factors such as software productivity, development technology, and interaction with the customer may vary substantially across different teams of developers even if they were working on building the same product. In fact, making software engineering processes, projects, and products reproducible is a challenge [Anda et al., 2008]. Therefore, when using a technology that may or may not be new to the people who have to work with it, it is important to consider the human factor and not just the objective data. Therefore, we believe that the subjective measures that we used enhance the reliability of our results more than any concerns about validity that may arise because of these measures.

4.6 Conclusions

As our results indicate, we do not see any of the technologies coming out on top for all the use cases. Instead, we must focus on the results from individual use cases in combination with the results from our subjective evaluation to judge each candidate database technology.

Python-CSV is clearly the best technology for UC I-1 in all respects because it will necessarily always require zero time to perform. For UC I-2–UC I-8, we see that Python-CSV has poor scalability both in terms of nodes and edges, as compared to its best competitors. We therefore abandon Python-CSV from further consideration.

MySQL and PostgreSQL have comparable performance; however, PostgreSQL clearly outperforms MySQL in maintaining the graphs. MySQL is best in storing graphs for smaller graphs; PostgreSQL scales better in storing the graphs and outperforms MySQL for larger datasets.

As for Neo4j, it performed poorly in retrieving the model and had the worst performance of all technologies in storing the model. Nonetheless, we did find that both Neo4j and PostgreSQL exhibit near constant performance and excellent scalability for UC I-3–UC I-8. The base processing time of Neo4j tends to be less than PostgreSQL, which is more desirable for any use case. Thus, Neo4j is the better database technology to maintain existing data in the database. It is also the newest technology among the considered candidate database technologies which offers greater flexibility but weaker security and less level of support.

MySQL and PostgreSQL being the relational databases are highly mature, and offer a good level of support and security.

Based on our study, SDATs which have a higher demand to store/retrieve new models would find PostgreSQL to be the better fit. Whereas SDATs which are inclined towards analyzing a limited quantity of software at a given time and involving high maintenance of the models in the database would find Neo4j as the most suitable option.

We thus conclude that PostgreSQL and Neo4j are the most likely candidates for an SDAT for which scalability and the full slate of our use cases are desirable. It remains to be seen whether this conclusion is maintained when the costs accruing from a database connector are factored in.

4.7 Summary

We described in detail the study as we conducted it to measure the performance of database technologies in order to analyze any potential for using them in SDATs. We presented and discussed the setup for evaluation and our results. We found that Python-CSV does not scale well in implementing the use cases; with an increase in the size and complexity of the data, MySQL exhibits mediocre performance. Neo4j performed the best in delete and change operations and PostgreSQL performed the best in store and retrieve operations over realistic analysis graphs; both of these remain viable options as technology choices to address the needs of SDATs to store and retrieve their models.

Chapter 5

Study II: Integrating Database Technologies in ModCP

In this chapter, we implement and integrate PostgreSQL database technology and Neo4j database technology in ModCP, as being the most promising candidates from Study I. We address a critical factor which the past work in the area has not addressed properly, i.e., the additional cost of connecting database technology to concrete software and comparing the costs of each thereafter. We discuss our approach for connecting the database technologies. We define a set of use cases that would be pertinent to ModCP and evaluate the performance of the database technologies in conducting these use cases for six candidate datasets. The purpose of this study is to address the following research question:

> **RQ2**: *What is the impact of adding/integrating datastore technology support to an SDAT on the re-computation cost of the analysis model?*

We find that the cost of re-computation increased dramatically when the database connectors of the database technologies (Neo4j and PostgreSQL) were used to store and retrieve the model. We used the same measures of performance that we utilize in Section 4.1. Section 5.2 discusses the six datasets we used for this study. The study setup is explained in Section 5.3. We present the results and our analysis thereof in Section 5.4. Remaining points for discussion, relative to this chapter, are given in Section 5.5. Section 5.6 presents our conclusions in response to the research question.

5.1 Performance measures

Similar to Study I, we compare the time consumed in responding to selected queries as objective measures and the difficulty of installation and ease of use as subjective measures of the database technologies to evaluate in this study.

5.2 Experimental dataset

For our study, we chose six open-source projects as input for the ModCP tool to account for variety in sizes and level of development and maintenance history. The same datasets were used to evaluate the ModCP tool in previous work Men [2018]; therefore, they provide a fair comparison between the old and new rebuild times. The information on these datasets is mentioned below and the specifics are described in Table 5.1.

1. **JCommander** is a command line annotation based parsing framework for Java 8.
2. **JSON-Schema-Validator** (JSV) provides validation support for the JSON Schema draft specification.
3. **Ant-Ivy** is an open source project dependency manager used to manage project dependencies by identifying, resolving, and informing them to the user.
4. **Apache Maven** is software project management and comprehension tool used to manage building, reporting and documentation of a project.
5. **Apache JMeter** is an open source application to measure performance and load test the behaviour of the application
6. **C/C++ Development Tooling** (CDT) is a project which provides an integrated development environment (IDE) for C/C++ within the standard Eclipse IDE.

Table 5.1: Information on dataset used as input project in ModCP to evaluate in the study.

Dataset	SHA	Statements
JCommander	f4b4df	1,747
JSV	c2ad15	2,689
Ant-Ivy	1f6a2a	9,789
Maven	5988d0	23,226
JMeter	708f98	59,632
CDT	d45ff1	592,710

["1,747", "2,689", "9,789", "23,226", "59,632", "592,710"]

5.3 Setup for evaluation

In this section, we describe the setup for our evaluation for analyzing the database technologies when implemented in ModCP including the use cases that were analyzed, and the database technologies used.

5.3.1 ModCP and the database setup

ModCP is written entirely in C# and the tool project was set-up on Microsoft Visual Studio Community 2015 Version 14.0.25431.01 Update 3 and Microsoft .NET Framework Version 4.8.04084. We used official connectors for PostgreSQL and Neo4j to connect to the database servers to establish the database connection. Npgsql was used to connect to PostgreSQL database server; it is an open-source ADO.NET data provider for C# projects, which is implemented in C#. Similarly, we used Neo4jDotNetDriver, another ADO.NET data provider to connect to Neo4j graph database server using the Bolt protocol.[1]

Figure 5.1 demonstrates how to connect to the database server.

1. ModCP, with the means of the PostgreSQL/Neo4j connector sends a connection request to database servers. The connection request from Npgsql is shown in Listing 5.1. A connection string is provided including *host, username, password* and *database*, and is used by the

[1] Bolt protocol. `https://neo4j.com/docs/java-reference/current/java-embedded/bolt/` [2021/06/29]

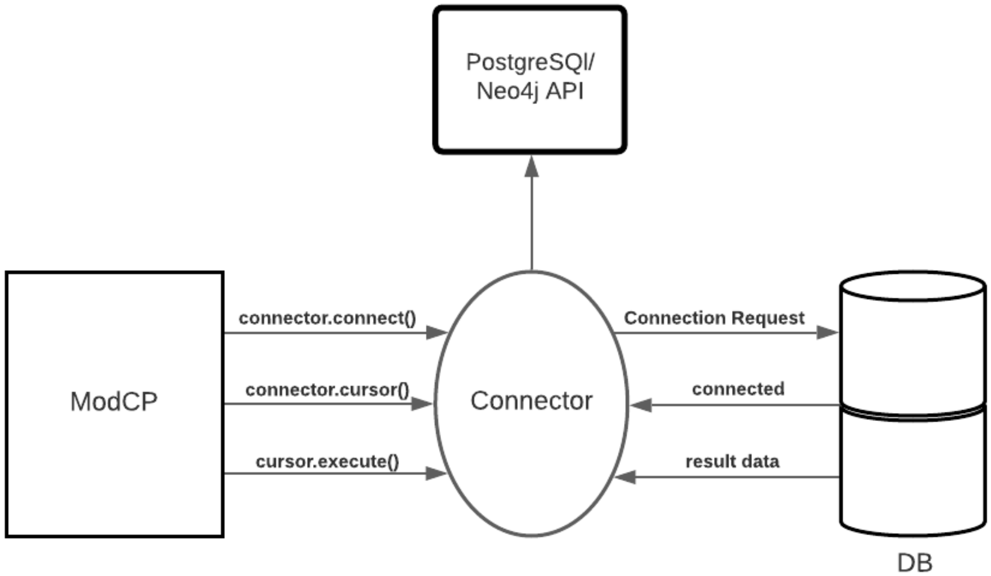


Figure 5.1: Connecting ModCP to the database server.

database server to authenticate the connection request. Similarly, the connection string of Neo4jDotNetDriver is shown in Listing 5.2 where *hostname, username* and *password* are provided.

2. Thereafter, the connection string credentials are verified and validated, and a secure connection is established between the database and ModCP. This connection can be accessed by a cursor, which is used to access or modify relations/graphs in the database.

3. A set of queries are applied on the cursor to store the model and load the data as per request. A load query retrieves the model data from the database and re-instate the model based on the loaded values.

The database connectors used in this study are briefly discussed below.

5.3.1.1 Npgsql

Npgsql is an open source ADO.NET data provider for PostgreSQL. It enables access to PostgreSQL database server to tools/software written in C#, Visual Basic, F#. Npgsql is written 100%

Listing 5.1: Connecting to PostgreSQL database

```
NpgsqlConnection conn =
    new NpgsqlConnection("Host=localhost;"\
        "Username=postgres;"\
        "Password=Str0ngPassword!;"\
        "Database=postgres");

conn.Open();

foreach (var node in nodes)
{
    NpgsqlCommand command =
        new NpgsqlCommand("insert into ecf_node values('" + node + "')", conn);
    command.ExecuteNonQuery();
}

conn.Close();
```

in C#; it is free and open source. The recommended installation is by using its NuGet package where all the latest versions updated regularly; we used version 4.0.2. The older, and now obsolete versions of Npgsql, came with Visual Studio extension (VSIX) that enabled connecting to PostgreSQL server within visual studio's explorer. PostgreSQL also offers database connectors for .Net, C, C++, Delphi, Java, JavaScript (Node.js), Perl, PHP, Python and Tcl.

5.3.1.2 Neo4jDotNetDriver

Neo4jDotNetDriver is a .NET driver used for connecting Neo4j databases using an in-house binary protocol, called Bolt. Its documentation is available on GitHub as well as on the official website of Neo4j. Neo4jDotNetDriver can be downloaded from NuGet; we used version 1.7.0. Neo4j also provides its official database connector drivers for Java, Spring, JavaScript, Go, and Python.

5.3.2 Use case definitions

We implement the three base use cases discussed in Study I, which are briefly discussed below.

UC II-1: *Create the model*. The model is encoded and stored on the database technology. It also involves the creation of schema for relational database.

Listing 5.2: Connecting to Neo4j database

```
IDriver driver = GraphDatabase.Driver("neo4j://localhost:7687", AuthTokens.Basic("username", "
    Str0ngPassword!"));
IAsyncSession session = driver.AsyncSession(o => o.WithDatabase("neo4j"));

try
{
    IResultCursor cursor = await session.RunAsync("CREATE (n) RETURN n");
    await cursor.ConsumeAsync();
}
finally
{
    await session.CloseAsync();
}

...
await driver.CloseAsync();
```

UC II-2: *Retrieve the model*. ModCP was restarted and the model was retrieved and re-instated.

UC II-3: *Update the model*. The modification made in the software being analyzed are reflected in the model. Updating a model involve deleting a class and modifying a method in the code of the software being analyzed

We used the same set of queries from Chapter 4 and ran them via the database connector of each technology.

5.3.3 Time measurement

The use cases were implemented on ModCP using candidate database technologies. Each use case implementation for each database technology was run ten times, and the space and time was recorded for each run. `stopwatch.ElapsedMilliseconds` was used to capture the execution time of the run in milliseconds. The space measurement was noted from the user interface of the database technology.

5.3.4 Time measurement

Each of these use cases were implemented on the candidate database technologies five times, and the time taken to process the query were recorded. The average query processing time to realize a use case was recorded to evaluate the database technologies.

5.4 Results and analysis

We describe and analyze the results from our objective and subjective evaluation for the databases in the following sections. Results are provided in terms of the measures used for each of the evaluations.

5.4.1 Objective evaluation

In this section, we discuss the results of this study conducted to evaluate the performance of the selected database technologies from Chapter 4. The complete results for this study are shown in Table 5.2. We found that the time taken to store the model for the considered datasets was lower for

Table 5.2: Execution times for the use cases, in seconds.

Dataset	UC II-1: Store		UC II-2: Load		UC II-3: Update	
	Neo4jDotNet...	**Npgsql**	**Neo4jDotNet...**	**Npgsql**	**Neo4jDotNet...**	**Npgsql**
JCommander	11,247.69	**9,872.43**	**1,024.52**	2,528.46	51.59	**44.35**
JSV	18,446.04	**17,024.50**	**1,897.73**	4,781.13	51.15	**47.95**
Ant-Ivy	77,388.98	**59,864.12**	**6,683.81**	11,981.13	53.67	**43.47**
Maven	172,800+	172,800+	—	—	—	—
JMeter	172,800+	172,800+	—	—	—	—
CDT	172,800+	172,800+	—	—	—	—

Table 5.3: The interim wait time (in seconds) to rebuild the ModCP model, rather than reload it.

Dataset	Time (s)
JCommander	5.240
JSON-JSV	5.814
Ant-Ivy	28.593
Maven	53.942
JMeter	137.712
CDT	2,290.048

Npgsql than for Neo4jDotNetDriver. Npgsql also outperformed Neo4jDotNetDriver in updating the model with the changes made in the software being analyzed; however, the difference was not significant. We observed that Neo4jDotNetDriver completed loading the model back in ModCP in approximately half the time taken by Npgsql. As shown in Table 5.2, the time involved in storing the largest three datasets using the two database connectors took over 48 hours; as this is unacceptably slow to meet practical limits, we deem that these technologies do not meet our needs and so we did not perform UC II-2 and UC II-3 on them.

Table 5.3 shows the time required to re-build the model of ModCP for the considered datasets, rather than reload it. Rebuilding costs a small fraction of the time relative to reloading the model (in the neighbourhood of 1% of the time); storing and reloading the models via such technologies renders them infeasible for realistic models.

5.4.2 Subjective evaluation

In this section, we present the results of our subjective evaluations. We discuss in detail the information that we gathered and how the database connectors under consideration perform for specific measures.

5.4.2.1 Maturity and level of support

Maturity and level of support can be considered to be huge indicators of success in the adoption of any new technology as it provides insight into stability, support for identifying and solving bugs, etc. PostgreSQL has had time to develop a dedicated user base and there exists extensive documentation for queries, bugs, and other issues that a developer might face. Neo4j, on the other hand, is fairly new in comparison and does not have the level of support that exists for PostgreSQL. PostgreSQL benefits from having SQL available as a unified language through which to interact with the database. Graph databases lack such a unified language and are also not as heavily adopted. During our implementation efforts, we often found it difficult to find help regarding Cypher, the query language of Neo4j. Most of the user support, limited as it is, comes from a wiki on the Neo4j website. Therefore, despite having a small active base, as compared to PostgreSQL, documentation is rather lacking.

5.4.2.2 Security

The connections via Npgsql to the database technology are established by a simple username and password, which is usually added in the code itself. There are alternate ways to provide the connection string by either a configuration file ("pg_hba.conf")which can be accessed by the Passfile function or a ProvidePasswordCallback can be used to generate a password in code. However, logging in with a username and password raises threats around secret management as the password must be shared with the application being used. Npgsql provides other means of authentication[2] for this purpose. Furthermore, encryption is disabled by default for PostgreSQL, but this can be easily changed to ensure secure connection.[3]

Similarly, Neo4jDotNetDriver also provides several authentication mechanisms including the basic authentication (i.e., username and password combination), Kerberos authentication (i.e., an authentication token with a base64 encoded server authentication ticket), and custom authentication (i.e., an authentication helper for when security provider has been built). These options do not stack up against the numerous options available for PostgreSQL. Neo4j supports encryption using standard SSL/TLS technology for client–server communication and also provides other options to secure communication channels.[4]

5.5 Discussion

In this section we discuss remaining issues relative to this study.

[2]PostgreSQL Authentication. `https://www.postgresql.org/docs/current/auth-methods.html` [2021/06/29]
[3]PostgreSQL Encryption. `https://www.postgresql.org/docs/current/ssl-tcp.html` [2021/06/29]
[4]Neo4j Security. `https://neo4j.com/docs/operations-manual/current/security/checklist/` [2021/06/29]

5.5.1 Space

As per the findings described in Section 4.5.1, we do not include the space required by the model within the database as there was no significant difference.

5.5.2 Alternative approaches

The candidate datastore technologies for our study were based on the results of Study I, but those results did not account for the required, additional cost of database connectors. While we found PostgreSQL and Neo4j to be promising database technologies, we recognize that other database technologies could have been valid as well. Oracle database is a multi-model database management system which is designed for enterprise deployments. It supports distributed databases and involves a licensing fee. Microsoft SQL Server, another relational database management system, is an official and inbuilt database for Microsoft Visual Studio. Visual Studio offers advanced in-built data editing, viewing and comparison of the schema options.

The core reason for pursuing database technology in our research was the that it was already at the heart of the operation of the industrial tool set (from Find it EZ[5]) into which ModCP will ultimately be integrated. Although valid database technology options exist, they are unlikely to perform significantly better: the database management system is not aimed at storing and retrieving the whole dataset frequently as the major use case, but to offer accessible selective data as per the request. Although we did not know this at the beginning, it is clear now that this be the case.

5.6 Conclusions

As our results indicate (Table 5.4), we do not see either of the two technologies performing acceptably for all the use cases. Therefore, we emphasize the results from individual use cases in combination with the results from our subjective evaluation to judge each candidate database technology.

[5]https://www.finditez.com/ [accessed 2021/06/24]

Table 5.4: Summary of performance (in seconds) of the integrated database technologies as compared to the baseline for ModCP without database support.

Dataset	UC II-1: Store		UC II-2: Load		Sum		Baseline
	Neo4jDot...	Npgsql	Neo4jDot...	Npgsql	Neo4jDot...	Npgsql	
JCommander	11,247.69	9,872.43	1,024.52	2,528.46	12,272.21	12,400.89	5,240
JSV	18,446.04	17,024.50	1,897.73	4,781.13	20,343.77	21,805.63	5,815
Ant-Ivy	77,388.98	59,864.12	6,683.81	11,981.13	84,072.79	71,845.25	28,593
Maven	172,800+	172,800+	—	—	?	?	53,942
JMeter	172,800+	172,800+	—	—	?	?	137,712
CDT	172,800+	172,800+	—	—	?	?	2,290,048

Storing the model on Neo4j using Neo4jDotNetDriver was expensive compared to storing it on PostgreSQL database using Npgsql. However, loading the model back in ModCP using Neo4j was faster than PostgreSQL. The cost of updating the model in the database was approximately the same (no significant difference) and constant for the smallest three datasets.

The PostgreSQL and its connector is more mature and offers better support than the newer non-relational database technology Neo4j. Both Npgsql and Neo4jDotNetDriver offer several similar authentication methods and options to encrypt the connection between the software and the database technology.

Based on our study, we found that integrating and storing the model in the database technology was unrealistic as the re-build cost far exceeded the traditional re-building time of ModCP.

We thus conclude that implementing a database technology and encoding and storing the SDAT model on it comes with excessively high overheads, leading such technologies to be impractical solutions to the performance problems of ModCP; simply rebuilding the model would be preferable. However, in later chapters, we consider an alternative kind of technology which we will find is of benefit.

5.7 Summary

We discussed in detail the study we performed to identify any increase in cost on integrating the database technologies to ModCP. We presented and discussed the setup for evaluation and our results. We found that the cost of storing the model was cheaper for PostgreSQL while the cost of retrieving the model back in ModCP was cheaper for Neo4j. Modifying and deleting the model in the database using connector was not significantly different between them. We realized that the impact of integrating a datastore technology using a connector comes with a high overhead and results in unacceptably slow performance. The traditional approach of simple re-computation of the model performs far better; however, we will investigate an alternative technology in later chapters that performs much better.

Chapter 6

Object Serialization to Address the Problem

In this chapter, we explore the scope of object serialization technologies to achieve a reduced model re-build time in SDATs (Section 6.1). In Section 6.2 we discuss the types of serializer, their compatibility, and their serialization formats: we use such points as the six relevant factors in choosing candidate serialization technology; we consider and discuss 14 object serialization techniques. We examine related work on serialization performance and scalability (Section 6.3), concluding that it is insufficient to help us determine which technologies will best suit our needs. We conclude that four of the technologies we examine are of sufficient promise to warrant more in-depth empirical study (Section 6.4).

6.1 Object serialization and its potential benefit in SDATs

Modern software applications are commonly developed and deployed independently as components with a mutual goal, by constantly communicating ideas and exchanging data between developers [Denaro and Mariani, 2005]. This exchange of data in SDATs could be achieved by capturing the entire state of the objects and re-loading them from the local storage system. This offline memoization where the computed details by the program are stored on local storage instead of volatile memory could be achieved, in principle, by a process called *serialization*. Serialization enables the conversion of the in-memory representation of a structured object into a platform-independent stream of bytes. This stream can be written into a file for storage or transmitted over a network for communication purposes. The reverse action of reconstructing the same object using the stream of bytes in the same or different environment is called *deserialization*. Most programming languages these days provide a built-in library for serialization and extensive support to personalize the process according to the needs of the user. Java and Microsoft .NET share a similar mechanism where, along with the serialized object, its referencing objects are serialized as per the requirement of the developer [Wigley et al., 2009]. For instance, *remote method invocation* (RMI) is an approach which enables an object to invoke a method on the same or remote component in distributed systems [Emmerich, 2000]. In the context of SDATs, the analysis data availability and its maintenance can be achieved by capturing the entire state of the objects and re-loading them from the local storage system.

6.2 Serialization approaches

When implementing a serialization mechanism in an object-oriented environment, the developer must decide between a number of trade-offs between ease of use and flexibility. The process can be automated to a large extent, provided you are given sufficient control over the process. For example, situations may arise where simple binary serialization is not sufficient, or there might be a specific reason to decide which fields in a class need to be serialized. This section explores such factors to determine the potential object serialization for ModCP.

We consider and discuss a range of object serialization technologies which could be potentially implemented in ModCP. We understand and evaluate them on the following six factors.

1. *Type*: Whether the object serialization technology is based on full object graph serializers or schema based serializers.

2. *Data Format*: The format in which object is translated when serialized.

3. *Private Members*: The ability of the serializer to serialize private members of a class/interface.

4. *Stream Support*: Streaming of the translated object is supported.

5. *Language Support*: Since ModCP is entirely written in C#, we check if the serialization technology supports the C# programming language.

6. *Graph Support*: whether the serializer can capture the information with regards to the inheritance. In other words, if the object serialization library exhibits the capability of understanding the parent–child relation and building the multi-level hierarchy of the objects.

6.2.1 Types of serializer

Every serialization library has a different course of action to capture the state of an object, although they still can be broadly categorized on user-end-interaction and serializer behaviour [Morschel et al., 2019]:

- **Full object graph serializers** (FOGS) automatically infer the schema of the object and its referenced classes to serialize them. In FOGS, the user has to introduce minimal changes to support changes, as it is easy to introduce and extend as the underlying software code base expands. FOGS, on the weaker side, allow few or no means of controlling the process of default serialization; they are known to generate large translated streams. The default procedure of FOGS is to serialize all the fields of the object as well as the full detail of the connected objects.

- **Schema based serializers** (SBS) obligate the user to explicitly mark the entities to be serialized which bestow control over the default behaviour of the serializer. *Pickling* is a process where the developer decorates the desired classes and fields explicitly, conveying to the serializer that only the mentioned entities are required to be captured. This activity allows the developer to hide the private details and at the same time administer the size of the serialized object. SBS mostly tend to be time- and space-efficient compared to FOGS because of the course of action supervised by the user. However, explicit schema declaration is intricate to introduce, and gets even more complex as new serializable classes need to be added. Also, while deserializing such streams, an extra measure to decode the schema might be required. SBS are more secure (than FOGS) if the primary use case is data interchange.

6.2.2 Serializer compatibility

Serialized objects follow a schema, that is, a defined structure described in a formal format supported by the serialization technique [Elmasri and Navathe, 1999]. This schema is independent of how the model data is encoded in the codebase. The class whose object is being serialized may change over time during its maintenance and this may result in inconsistency between the object being serialized in the past and the current structure of such objects' state [Sud et al., 2020].

1. **Forward compatibility** of a serializer means that the developer with the older schema can still parse and serialize the current/future versions of the software.

2. **Backward compatibility** of a serializer means that a developer with the current schema decoration can (de)serialize current and past versions of the software. It prevents data loss as the serialized data may not be used to load the model back in the newer SDAT software version.

6.2.3 Serialization format

The space efficiency of the translated serialized object is one of the important factors for serialization techniques. As highlighted by Sumaray and Makki [2012], the storage and data interchange formats have evolved from markup languages to structure oriented encoding of information; identifying the best storage format has been a complex topic for research, with inconclusive results. The data formats can be broadly categorized as text-based (human readable) and binary formats. Although the data formats are categorized on the basis of human readability, any non-trivial computation and analysis of large data requires software assistance and so human readability holds no special value in our consideration of serialization technologies. Below are some common serialization output data formats:

6.2.3.1 Text-based

Text-based storage formats are human-legible ones that mostly use an organized tree structure, offer good readability and maintainability. They require large space and parsing of each character, thereby leading to performance impacts on deserialization.

1. **Extensible Markup Language (XML)** is a text-based serialization format used to store information in structured form [Bray et al., 1997]. It was developed as easy-to-use language over internet and is extensively used in other applications such as storing documents, transaction, books etc. It is known to be one of the most commonly used formats for storing and sharing information in distributed systems, remote procedure calls and object serialization [Nurseitov et al., 2009].

2. **JavaScript Object Notation (JSON)** is a lightweight alternative to XML which is built on name/value pairs and an ordered list of values. It is language-independent and thus widely used as a data-interchange format. A point to note is that the name in a name/value pair can only be a string and it does not support any date format. Many companies have recently started their web services in JSON and it is even widely accepted in mobile devices over XML [Sumaray and Makki, 2012]. JSON's structure resembles common objects and data structures of various programming languages, which makes it a good option for our scenario. It does not support comments but these are not required in our context anyhow, when storing large amounts of analysis model information (if comments are really required, they could be added as a name/value pair). JSON was initially developed to be used with the JavaScript programming language and thus it saved a lot of overhead compared to XML. JSON is parsed and generated faster in machines when compared to similar storage formats; however, it lacks namespace support, measures to validate input, and extensibility features [Nurseitov et al., 2009]. The syntactical structure of JSON is human readable.

3. **YAML Ain't Markup Language (YAML)** is an open source text-based data storage and interchange format [Ben-Kiki et al., 2008]. It is a compact and human-legible serializa-

tion format that is easy to generate and manipulate. YAML incorporates strict indentation from Python and a brief format for data structures; thus, it is also considered a superset of JSON.YAML cannot only encode native data types such as strings, integers, floats, lists, etc. but it also allows the user to define their own data types to store; this is useful in the application of object serialization.

6.2.3.2 Binary format

Binary file formats store values in typically compact native formats, requiring no parsing/translation to interpret the data stored within [McGrath, 2003].

1. **MessagePack** is a binary object serialization format which offers advantages over text-based in terms of space efficiency. It works on the concepts of type system and formats: using the type system to convert objects into MessagePack format and to re-convert MessagePack format back to objects. It is also known for the small encoding size of the data types over JSON, thus serving as a good choice where space efficiency is a priority. One of the major flaws of MessagePack is that it does not perform well in client side serialization [Furuhashi, 2013]. MessagePack has a limited range of values for integers, binary objects, string objects, number of elements in array objects, and key/value pairs in map objects.

2. **Protocol Buffer (Protobuf)** is a compact serialization format developed by Google to be used internally for serialization, which was later released as open source. Protobuf is a binary serialization storage format which was designed in comparison to XML for compactness and better performance in communicating across networks, or to be used as a storage medium locally. It is platform-independent and is supported by many programming languages such as Java, Python, C++, C#, Ruby, and JavaNano. It is primarily used in object serialization, remote method invocation and simply interchanging information. Protobuf allows the user to describe and structure the information to store, and thereafter the same structure can be used to store the same or similar information. It provides extensive support for data types, more so than JSON, aiding functions and methods to work with these. Each message type in

a .proto file is a name/value pair, where the value can be numbers (integers or floating-point numbers), truth values, strings, raw bytes, or even other protocol buffer message types that allow the user to parse the data hierarchically.

6.2.4 Object serialization technologies

The following sections describe the robust serialization technologies considered in our subsequent studies.

6.2.4.1 Binary serialization

BinaryFormatter is a native .NET deep serializer found in the System.Runtime.Serialization library. It is a FOGS, that stores all the information for a class, including class member information and class metadata along with the attribute structure and the assembly of the class. As with FOGS, implementing binary serialization as a storage solution is simple: the class to be serialized is marked with a Serializable attribute. The Serializable attribute is not passed down to the inherited classes, so those must be marked as well if they are also to be serialized. BinaryFormatter is also capable of handling cyclic references of the object graph. Even though BinaryFormatter is a FOGS, the class members that possess private or sensitive data can be marked not to be serialized.

6.2.4.2 XML serialization

XML serialization is a native SBS that allows control over the encoding of objects into XML through System.Xml.Serialization class provided by .NET. To control the XML generated from serialization, classes and fields need to explicitly marked with serialization attributes. XML serialization does not allow storing or transmitting the private fields of the class. It also does not provide object graph support. It uses text-based serialization format and does not provide object graph support.

6.2.4.3 DataContract serialization

The DataContract serializer is derived from the XmlObjectSerializer, which translates the object to XML and back. DataContract is an SBS, which requires classes and fields to be marked with DataContract attributes to include them in the serialized stream of object. Inheritance needs to be specified explicitly by decorating the parent child classes with "knowntype" attributes. Classes marked with DataContract attributes can also use the Protobuf serializer and moreover it can also transform the object into JSON format. DataContract serializer allows one to store/transport public as well as private fields unlike XmlSerializer.

6.2.4.4 MessagePack

MessagePack is an object serialization specification which translates objects into binary streams. MessagePack, an SBS, requires marking the desired class and class members to be serialized explicitly. MessagePack offers serializing private class members too. It provides graph support by default and provides support for the C# language.

6.2.4.5 Protobuf-net

Protobuf-net is an open-source contract-based serializer available on GitHub.[1] It uses Google's protocol buffer format as the translated output of the serialization process and it uses a 32-bit binary format. It is an SBS, so it requires the developer to decorate the classes and class members to be serialized. Protobuf-net allows private members to be serialized. Any inheritance must be explicitly specified, using the ProtoInclude attribute, which allows the serializer to recognize the child class. Cycles and references are disabled by default to lower the cost; however, these are supported and can be turned on manually.

[1] Protobuf-net. `https://github.com/protobuf-net/protobuf-net` [2021/06/29]

6.2.4.6 FlatBuffer

FlatBuffer is an open source, cross-platform object serialization library. As the name of the approach suggests, FlatBuffer uses flat buffers as the serialization data format. It is an SBS, which means that the desired class and class members need to be marked in order to be captured during the serialization process. FlatBuffer can serialize private class members and it also offers full graph support.

6.2.4.7 NewtonSoft Json.NET

NewtonSoft Json.NET permits an object to be serialized in JSON format by applying three different implementations: JavaScriptSerializer, DataContractJsonSerializer, and Newtonsoft-net. Json-.NET is an SBS, that enables the user to mark the desired class members to serialize. Json.NET provides the capability of capturing private and public fields both. Much like DataContract and Protobuf-net, classes and fields needs to be decorated with the serialization attribute in order to include the details in the JSON output produced. Inheritance must be explicitly specified in the model in order to re-instate it at the time of deserialization. NewtonSoft Json.NET exhibits the graph support capability, that is, it can handle cyclic references of members, thereby avoiding the redundant translation of the same class member.

6.2.4.8 JIL

JIL is an open source fast JSON (de)serializer built atop of Sigil. It belongs to FOGS, and uses the text-based JSON serialization format. Inheritance in JIL must be explicitly specified. It only supports serializing public class members and does not provide graph support.

6.3 Prior work on performance and scalability of serialization

Hericko et al. [2003] compare binary serialization and XML object serialization on Java and Microsoft .NET, to identify the efficiency and differences in the performance in terms of time and size taken to serialize and deserialize the objects. The authors serialized 100, 1,000, 10,000, and 100,000 objects in four different test runs where each object had the same size and complexity. They average the results of the serialization process. They observed that the size of the serialized objects was more compact in Java as compared to Microsoft .NET. Microsoft .NET headers include details about the assembly, its location, culture, and the optional public key used for integrity purposes resulting in its larger serialized object size. Due to the results being averaged, it does not truly represent real world possibilities. Their results show a linear growth for the size of the serialized object against the number of objects. The flaw seems to lie in the fact that the size and complexity of the object to be serialized is not taken into consideration in their work. Scalability of these serialization techniques cannot be assessed using simple objects. The study involves (de)serializing the same objects several times to address the scalability which does not take other member types and complexity into consideration. Another thing to note is that the XML serializer may not accurately capture the objects in a real software due its failure to serialize private attributes. A real-world software code base aspires to incorporate the strongest form of encapsulation; this study fails to address that.

Kono and Masuda [2000] propose a serialization mechanism to reduce serialization overhead at both ends of the remote method invocation (RMI) communication. They compare their approach with existing serialization routines. Their goal is to simplify the process of RMI by using runtime knowledge about the receiver's platform, and redefining the serialization routine accordingly. The object to be serialized on the sender side is directly converted into the receiver's in-memory representation; this allows the receiver to access it immediately without any data copy and conversion which they present as the cause of large overhead. To generate a specialized routine, they used the Tempo Specializer,[2] a partial evaluator for C programs. They compared the performance of three serializing routines: XDR,[3] noDS, and DS. XDR is a conventional serializing routine that uses the Sun XDR library and noDS is a generic (not specialized) serializing routine. DS is the serializing routine dynamically specialized by the Tempo Specializer. They compared the serialization

[2]Tempo Specializer. `http://phoenix.inria.fr/software/past-projects/tempo/manual.html` [2021/06/29]
[3]XDR. `https://tools.ietf.org/html/rfc1014` [2021/06/29]

time and space to evaluate the three candidate libraries. Their results show that they achieved a 1.9–3.0 times improvement over the Sun XDR serialization library. The noDS and DS routines also show better performance than XDR in both homogeneous and heterogeneous environments. They also show that noDS was 2.0 times as fast as XDR and in the heterogeneous environment 1.35 times as fast as XDR. However, not enough information has been provided about how the experiment was conducted in order to reproduce it or to draw conclusions from it relevant to our context. In addition, the comparison of serialization routines is specific to RMI and not serialization techniques in general because RMI involves communication of serialized files over networks. So, space optimized serialization techniques will be preferred. Whereas, in our scenario, space has less preference and time optimized technique is of high importance. Also, our use case may or may not involve sending/receiving translated object over network. Another major difference in the use of RMI and of SDAT checkpoint/restart is that RMI involves marshalling and our use case does not. In other words, in RMI, the source code of client side is also sent along the translated object. Whereas, in our case, the client and server sides are usually the same: the object data is serialized and deserialized on the same platform/machine. However, even if this were not the case, the source code will always be the same; thus, there should be no need to marshal the object.

Sumaray and Makki [2012] address the need for an efficient serialization storage format in mobile devices. The authors compare speed, space, and usability of four widely accepted serialization formats: XML, Protobuf, JSON, and Thrift, by way of an Android application they developed; they used two test objects: book and video. We observed that the source code provided by the authors for the book and video classes comprised only primitive datatypes, such as **int**, **double**, **bool**, **string**, etc., which are straightforward to translate and reinstate. Also, there were no object references and inheritance in the dataset source code, which is a core aspect of object-oriented programming. In addition to time and space measures for the serialization process, the authors considered the ease of implementation and documentation of the serialization libraries. The study concluded that binary formats tend to be space- and time-efficient, thus being preferable to traditional text serialization formats. Both Protobuf and Thrift performed about the same, whereas JSON outperformed

XML marginally in serialization and deserialization times, and substantially in space taken by the translated object. Regardless of the standard evaluation parameters used to compare serialization formats, the results drawn from the study are not reliable because the dataset used in the study lacks real world complexity. Also, XML serializer, one of the compared serialization technologies, is not applicable to our context, as it does not allow the user to serialize private attributes. This leaves us with the need to implement serialization in real-world SDATs in order to evaluate available serialization technologies and formats.

Andrade et al. [2015] and Kono and Masuda [2000] compare different serialization techniques; they also propose a new technique and benchmark them by comparing with other formats. In Andrade et al. [2015], a systematic evaluation was performed of their own approach against existing data marshalling approaches such as Google's Protobuf and Lightweight Communication and Marshalling (LCM). They studied varying numbers of integral and double data fields to compare the data sizes of the three protocols and studied the application of serialization in distributed systems such as self driving vehicle application. In Giaimo et al. [2015], they showed that the bandwidth consumption by the data being exchanged can be saved between two consecutive packets by only sending the difference. Their results suggest that Google Protobuf is a better option when all integer fields are involved whereas their own self-adaptive marshalling approach performs better when a large number of non-integral fields are involves. LCM was found to send more compact data when the number of message fields was increased. Andrade et al. [2015] do not provide any other heuristic of performance evaluation and rely solely on the size of the packets for their results. Although, our use case is object serialization, but the concept applied here is similar. However, only primitive data types such as double and integer data types are serialized in the study and this does not help us answer our large software scalability problem.

Although not a comparative study, Tauro et al. [2012] describes how binary object serialization is performed in .NET. does not evaluate it on any software or tool. Binary serialization is also discussed for C++ and and they do not compare it with other possible binary-format serialization approaches. There is no comparison of the performance of same serialization approach on different programming language has been done. This work may be used as an introduction guide or "unofficial" documentation rather than as a performance comparison guide.

Maeda [2011] compares six serialization libraries: Java serialization, interface description language (IDL), XStream, Protocol Buffers, Apache Avro, and MessagePack. Java serialization is Java's built-in mechanism for manipulating objects as streams of bytes, IDL is a notation to define structured data. XStream is a Java-specific library to serialize and deserialize objects in XML. They repeatedly serialize small Java objects of a class containing simple data members and one *List* member, thousands of times. The serialization techniques are not implemented as a storage or communication technology for a software or tool, but to serialize a simple object in general. Therefore, it only takes the complexity of the data being serialized into account and not the complexity of the model of the application.

Maeda [2012], continuing the study from Maeda [2011], compared 12 serialization libraries in XML, JSON, and binary formats. Based on their results, in terms of the size of binary-based serialized data, the performance is better than XML-based and JSON-based serialization. For easy interoperability with dynamic languages, they suggest that Apache Avro and Protostuff (a Java serialization library) were the best in the twelve serialization libraries. The results provided display a linear relation between the file size of the serialized output vs. the number of objects which reflects that the number of objects have been increased without changing the complexity of it. In their evaluation they repeatedly serialize the same object instead of considering different types of objects of varying complexities. Real-world software grows constantly and the analysis model complexity increases exponentially and drastically. The lack of proper datasets to compare serialization libraries does not help in evaluating their actual performance when integrated to serve as a storage implementation for software analysis models. Some serialization libraries cannot keep track of cyclic references while other cannot serialize private members. Here, in the evaluation, a generic example has used on all 12 serialization libraries with just primitive data types fields.

Piisalo [2016] compares the efficiency and speed of different serialization formats. While there is a special focus on the Protobuf-net format which they ultimately choose as the right choice for their client company's game, other comparisons include the binary format of the Unity3D game development platform, Newtonsoft JSON, and XML. They wrote the performance tests of these serialization formats in C# and used Unity3D for evaluation. They measured the speed by computing the time taken for serialization and deserialization of formats, and efficiency by how large the file size was with the serialized data when saved to disk. Their results, based on two measures only, indicate that Protobuf-net format was the fastest among all formats while XML was the slowest. Protobuf-net also has the smallest size for serialized files while Unity3D binary format had the largest. The study fails to contribute any helpful insight in deciding on a serialization technique related to our work; the datasets considered in the study only consider primitive types for data members; the same object is serialized multiple times which does not address the complexity of real world data and thus the scalability; no referential cycles are present in the objects being serialized; and serialization techniques like XML serialization cannot serialize private field members which are required in our considered SDAT.

In an attempt to compare serialization formats and libraries across different languages, Vanura and Kriz [2018] performed a study with a focus on processing time and size of the output data. The formats that they compared included XML, JSON, MessagePack, Avro, Protocol Buffers, and native serialization of PHP, Java, and JavaScript. A total of 49 native and third party libraries were evaluated by using data types such as integers, decimals, logical values, strings, objects, and arrays/collections. The data members of objects do not contain complex objects and arrays and collections exist only of the types mentioned. The same data is repeatedly (de)serialized a thousand times for each library and the total time is measured. They repeat this process a hundred times, adding up to a total of 100,000 iterations for the same data. Google's ProtoBuf library implemented in Java provided the best efficiency based on their results. JSON format performed well among languages while XML libraries had the worst results. They concluded that the performance of different formats and libraries varied based on the language in which it was implemented and that a single format could not be accepted as the best performing across all platforms.

6.4 Conclusions

Table 6.1 summarizes the features and capabilities of the serialization technologies we have examined above, relative to the properties relevant to our context as described earlier (type, data format, private members, stream support, language support, and graph support) for an object serialization technology to be used within ModCP to improve the traditional re-build time. We seek serializers with support for C# language, serialization of private fields, and stream and graph support; based on these attributes, we select BinaryFormatter, DataContract, NewtonSoft JSON.NET, and Protobuf-net for deeper study: we will integrate these serialization technologies in ModCP and evaluate the performance of them against each other and the traditional approach of recomputing in ModCP, in the subsequent chapters.

Table 6.1: Summary of the object serialization technologies we studied.

Serializer	Type	Data format	Private members	Stream support	C# support	Graph support	Schema
BinaryFormatter	Graph	Binary	Yes	Yes	Yes	Yes	FOGS
DataContract	Tree	XML	Yes	Yes	Yes	Yes	SBS
FlatBuffer	Tree	Binary	Yes	Yes	Yes	No	SBS
JIL	Tree	JSON	No	Yes	Yes	No	FOGS
MessagePack	Graph	Binary	Yes	Yes	Yes	No	SBS
NewtonSoft Json.NET	Tree	JSON	Yes	Yes	Yes	Yes	SBS
Protobuf-net	Tree	Binary	Yes	Yes	Yes	Yes	SBS
XML Serializer	Tree	XML	No	Yes	Yes	No	SBS

6.5 Summary

In this chapter, we described various object serialization technologies with their advantages and disadvantages if implemented in SDATs. We discussed the different types of object serializers available, their compatibility, the different supported formats of the serialized object, and the serialization technologies which we consider for our work. We explored the prior work in this area and discussed it. We concluded that four serialization technologies (BinaryFormatter, DataContract, NewtonSoft JSON.NET, and Protobuf-net) are sufficiently promising in addressing our problem that we selected them for further study in later chapters.

Chapter 7

Study III: Object Serialization Technologies in ModCP

In this chapter, we compare several object serialization technologies on specific factors relevant to SDATs. The past work in this area does not suffice to determine an optimal technology for our work; for example: the serialized objects used in existing studies contain few fields and often only with primitive data types such as `int` and `string`; the data in the serialized objects lack non-trivial structure and do not represent real-world software data size and complexity; or the compared technologies are not used to capture the state of an actual SDAT.

In our study, we consider BinaryFormatter, DataContract, NewtonSoft JSON.NET, and Protobuf-net as the candidate object serialization technologies arising from our analysis in Chapter 6. We define a set of use cases pertinent to SDATs and evaluate the performance of the serialization technologies in conducting these use cases for datasets of varying sizes when integrated with a specific SDAT, ModCP. The purpose of this study is to address the following research question:

> **RQ3**: *How do object serialization technologies compare in terms of space and time cost required for model data storage and access mechanism when integrated with ModCP?*

We find that the DataContract serializer yields the best performance and reduced re-computation time.

We describe the measures of performance that we utilize in Section 7.1. We used the same datasets as in Section 4.2 as the experimental data for this study. The study setup is explained in Section 7.2. We present the results and our analysis thereof in Section 7.3. Remaining points for discussion, relative to this chapter, are given in Section 7.4. Section 7.5 presents our conclusions in response to the research question.

7.1 Performance measures

We utilize objective and subjective measures to evaluate in this experiment. Objective measures consist of the time taken to (de)serialize and the space required to store the model data using the serialization technologies. The subjective measures consist of the difficulty of implementation of a serialization technology, and the documentation and support offered by the provider. These measures were obtained by making use of the documentation available as well as the personal experience of the researcher, open source community work, and published papers.

7.2 Setup for evaluation

We implemented each of the four serialization technologies by decorating the classes of ModCP to enable the serialization process by the candidate serialization technology, one at a time. Decoration is a procedure where a serialization attribute is written at the start of each (desired) class or field declaration within the source code, subsequently used as a marker by the serialization technique to identify the target to serialize. All our candidate serialization techniques are schema-based serializers except the BinaryFormatter which is a full-object graph serializer. A total of 311 fields in 112 classes were decorated for serialization using the appropriate serialization attributes to be captured and stored in the translated serialized object. These were all the fields present in the classes implementing the models for ModCP (i.e., the extended call graph, the type hierarchy, and the modularized program dependence graphs).

7.2.1 Procedure

The computed model of ModCP can be translated and stored using object serialization technology. The implementation design is described in Figure 7.1. The end user uploads the software codebase for which the analysis is required into ModCP and the model is generated therein, which includes the CHG, ECG, and mPDGs for the software under analysis. The ModCP model is then serialized to a file on the local disk; the detailed serialization format is specific to the serialization technology. The computed model which is present in the memory is used to load the view of the ModCP tool, so the user can perform the required analysis. However, when the model is not available in the memory, the ModCP model is re-built using the serialized object file. Figure 7.2 shows the working of ModCP where the model is loaded from the serialized file without rebuilding it, as would occur in the traditional way. Once the model data is retrieved, the ModCP view is made available to the end-user to begin/resume their analysis of the software. load store - Store

7.2.2 Implementation

In this section we describe the implementation details specific to each candidate serialization technology.

7.2.2.1 BinaryFormatter

BinaryFormatter is the only FOGS amongst our candidate serialization technologies; we used version 4.0.30319. We added the `[System.Serializable]` attribute on the required classes to capture the ModCP model as shown in Listing 7.1. There is no need to mark fields explicitly, as BinaryFormatter captures the values by itself. However, if any class field is not desired to be serialized, e.g., in cases where hiding private or sensitive data may be needed, the `[NonSerialized]` attribute can be used to indicate that the element should not be serialized. The process of serializing the desired object using BinaryFormatter and writing the translated byte stream to a file on local system is done using the `Serialize(..)` function, as shown in Listing 7.2. This function takes two input arguments: an instantiated stream to which the object is to be serialized, and the object to be serialized. The

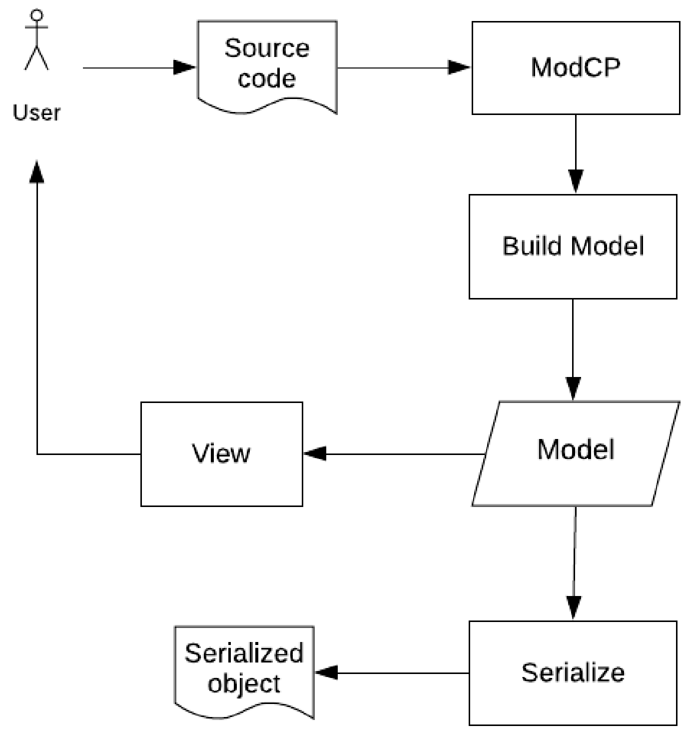


Figure 7.1: The process of storing the ModCP model using serialization technology.

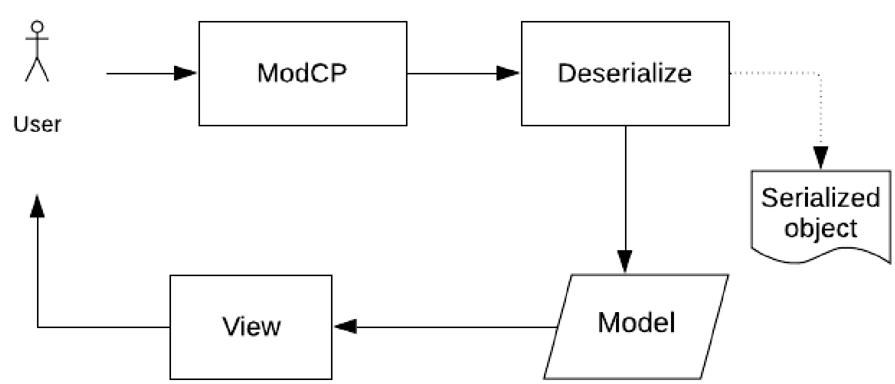


Figure 7.2: The process of loading the ModCP model back using serialization technology.

Listing 7.1: Decorating the classes to serialize using BinaryFormatter.

```
using System;

[Serializable]
public class MethodSignature
{
    public readonly string TypeQualifiedName;

    [NonSerialized]
    public readonly string Name;
    ...
}
```

Listing 7.2: Serialization process using BinaryFormatter.

```
public static void SerializeModel(PdgProject prj, FileInfo f)
{
    try
    {
        using (FileStream s = f.Open(FileMode.Create, FileAccess.ReadWrite, FileShare.ReadWrite))
        {
            serializer = new BinaryFormatter();
            serializer.Serialize(s, prj);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine("Failed to serialize. Reason: " + e.Message);
        throw;
    }
}
```

object is transformed into a byte stream followed by updating of the stream object. The stored file can be used to reinstate the object using the Deserialize(..) function which takes the encoded stream as an input, as shown in Listing 7.3.

7.2.2.2 DataContract

The DataContract serializer is an SBS, so we decorated the required relevant classes and the fields of the ModCP model, using version 4.0.30319; we used the [DataContract] attribute on classes and the [DataMember] attribute on fields. By default the DataContract serializer will serialize all objects it encounters by value. We preserved the object reference data by applying the option

Listing 7.3: Deserialization process using BinaryFormatter.

```
public static PdgProject DeserializeModel(FileInfo f)
{
    try
    {
        using (FileStream s = f.Open(FileMode.Open, FileAccess.ReadWrite, FileShare.ReadWrite))
        {
            deserializer = new BinaryFormatter();
            PdgProject loaded_prj = (PdgProject)deserializer.Deserialize(s);
            return loaded_prj;
        }
    }
    catch (Exception e)
    {
        Console.WriteLine("Failed to serialize. Reason: " + e.Message);
        throw;
    }
}
```

Listing 7.4: Decorating the classes to serialize using the DataContract serializer.

```
using System.Runtime.Serialization;

[DataContract(IsReference=true)]
[KnownType(typeof(CallSite))]
public class MethodSignature
{
    [DataMember(Order = 1)]
    public readonly string TypeQualifiedName;

    [IgnoreDataMember]
    public readonly string Name;
    .
    .
    .
}
```

IsReference=**true** on [DataMember] as shown in Listing 7.4 to mitigate the redundant serialization of the same object. The [DataContract] attribute is not passed down to its child classes, so the child classes need to be marked with [DataContract] attribute too when they are to be serialized. In order to provide knowledge of the inherited class members to the serializer, we need to decorate the parent class with [KnownType(typeOf(x))], where "x" is the child class. The [IgnoreDataMember] attribute can be used to indicate to the serializer to ignore a class field during serialization. Listing 7.5 shows the process of serializing the ModCP model and writing the serialized object in an XML

Listing 7.5: Serialization process using the DataContract serializer.

```
public static void SerializeModel(PdgProject prj, FileInfo f)
{
    try
    {
        using (FileStream s = f.Open(FileMode.Create, FileAccess.ReadWrite, FileShare.ReadWrite))
        {
            DataContractSerializer serializer = new DataContractSerializer(typeof(PdgProject));
                serializer.WriteObject(s, prj);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine("Failed to serialize. Reason: " + e.Message);
        throw;
    }
}
```

Listing 7.6: Deserialization process using the DataContract serializer.

```
public static PdgProject DeserializeModel(FileInfo f)
{
    try
    {
        using (FileStream s = f.Open(FileMode.Open, bFileAccess.ReadWrite, FileShare.ReadWrite))
        {
            DataContractSerializer deserializer = new DataContractSerializer(typeof(PdgProject));
            PdgProject loaded_prj = (PdgProject)deserializer.ReadObject(s);
            return loaded_prj;
        }
    }
    catch (Exception e)
    {
        Console.WriteLine("Failed to serialize. Reason: " + e.Message);
        throw;
    }
}
```

file. Similarly, Listing 7.6 shows the process of reading the serialized file and deserializing it to reinstate the ModCP model.

7.2.2.3 Protobuf-net

As shown in Listing 7.7, we decorated the relevant classes and fields of the ModCP model, using Protobuf-net version 2.4.0; we used the [ProtoContract] attribute on the classes and [ProtoMember] on

Listing 7.7: Decorating the classes to serialize using Protobuf-net.

```
using ProtoBuf;

[ProtoContract(AsReferenceDefault = true, UseProtoMembersOnly = true, SkipConstructor = true)]
[ProtoInclude(100, typeof(CallSite))]
public class MethodSignature
{
    [ProtoMember(1)]
    public readonly string TypeQualifiedName;

    [ProtoIgnore]
    public readonly string Name;
    ...
}


[ProtoContract(ImplicitFields = ImplicitFields.AllFields)]
[ProtoInclude(100, typeof(CallSite))]
public class ClassDeclaration : TypeDeclaration
{
    private const string Implicit = "implicit";
    public ClassOrInterfaceType ParentClass { get; set; }
    ...
}
```

Listing 7.8: Serialization process using Protobuf-net.

```
public static void SerializeModel(PdgProject prj)
{
    try
    {
        using (var file = File.Create("C:\\dataset\\model.bin"))
        {
            ProtoBuf.Serializer.Serialize(file, prj);
        }
    }
    catch (Exception e)
    {
        Console.WriteLine("Failed to serialize. Reason: " + e.Message);
        throw;
    }
}
```

the fields. We customized the [ProtoContract] attribute on each class; the option AsReferenceDefault=**true** was used to inform the serializer to serialize the class as a reference where needed, and the option UseProtoMembersOnly=**true** was used to serialize only the fields marked with the [ProtoMember] attribute. The constructor is always called by default at deserialization and an exception is thrown if

Listing 7.9: Serialization process using Protobuf-net.

```
public static PdgProject DeserializeModel(PdgProject prj)
{
    try
    {
        using (var file = File.OpenRead("C:\\dataset\\test.bin"))
        {
            PdgProject loaded_prj = ProtoBuf.Serializer.Deserialize<PdgProject>(file);
            return loaded_prj;
        }
    }
    catch (Exception e)
    {
        Console.WriteLine("Failed to serialize. Reason: " + e.Message);
        throw;
    }
}
```

no constructor exists. To skip calling the constructor, the option (SkipConstructor= **true**) can be used in the [ProtoContract] attribute. The ImplicitFields option may be used to automate the serialization process without individually decorating the fields within a class. ImplicitFields provides three options as to what to serialize: all the fields, no fields at all, or just the public fields. It must be noted that the options SkipConstructor and ImplicitFields applied to a [ProtoContract] attribute are never inherited; the developer needs to decorate the necessary elements with these attributes manually. [ProtoIgnore] can be used as an attribute for elements that contain sensitive or irrelevant information that the developer does not want to serialize/deserialize. [ProtoInclude(typeOf(x)], where "x" is the child class, is used to inform the serializer regarding parent–child class relationships, so that inherited fields are available in subclasses upon (de)serialization. Listing 7.8 shows the process of serializing the ModCP model. The Serialize() function takes a ModCP model and a .bin file as input parameters, writing the transformed ModCP model to a binary file. Listing 7.9 shows the reinstating of ModCP model by deserializing the ModCP model from the serialized object stored in binary file.

7.2.2.4 NewtonSoft Json.NET

NewtonSoft Json.NET is an SBS; therefore, we decorated the required relevant classes and the fields of the ModCP model, using version 12.0.2. We used the [JsonObject] attribute on classes

Listing 7.10: Decorating the classes to serialize using NewtonSoft Json.NET.

```
using Newtonsoft.Json;

[JsonObject(IsReference = true)]
public class MethodSignature
{
    [JsonProperty]
    public readonly string TypeQualifiedName;

    [JsonIgnore]
    public readonly string Name;
    .
    .
    .
}
```

Listing 7.11: Serialization process using NewtonSoft Json.NET.

```
public static void SerializeModel(PdgProject prj)
{
    try
    {
        string jsonData = JsonConvert.SerializeObject(prj, new JsonSerializerSettings
            {
                TypeNameHandling = TypeNameHandling.All,
                PreserveReferencesHandling = PreserveReferencesHandling.All
            });
        File.WriteAllText(@"c://dataset//jcommanderSerialized.json", jsonData);
    }
    catch (Exception e)
    {
        Console.WriteLine("Failed to serialize. Reason: " + e.Message);
        throw;
    }
}
```

and the [JsonProperty] attribute on the fields. By default NewtonSoft Json.NET will serialize all objects it encounters, by value. The [JsonIgnore] attribute can be used to indicate tell the serializer to ignore a field from the serialization process. We preserved the object reference data by applying IsReference=**true** on [JsonObject], as shown in Listing 7.10. The serializer can also be informed to preserve object references at serialization. As shown in Listing 7.11, we customized the JsonSerializerSettings to preserve object references by setting PreserveReferencesHandling to All and informed the serializer to consider inheritance by also setting the TypeNameHandling to All. The

Listing 7.12: Deserialization process using NewtonSoft Json.NET.

```
public static PdgProject DeserializeModel(PdgProject prj)
{
    try
    {
        PdgProject loaded_prj = JsonConvert.DeserializeObject<PdgProject>(File.ReadAllText(@"
          jcommanderSerialized.json"), new JsonSerializerSettings
            {
                TypeNameHandling = TypeNameHandling.All,
                PreserveReferencesHandling = PreserveReferencesHandling.All
            });
        return loaded_prj;
    }
    catch (Exception e)
    {
        Console.WriteLine("Failed to serialize. Reason: " + e.Message);
        throw;
    }
}
```

SerializeObject() function takes the ModCP model and the custom serializer settings as input parameters, and returns a serialized object stream in JSON format. This JSON string is then written to a file. Listing 7.12 shows the process of reading the serialized model from the JSON file and deserializing it to reinstate the ModCP model object.

7.2.3 Use cases

Object serialization technologies do not support modification of the serialized object; therefore, the identified use cases for this study are limited to:

UC III-1: *Store the model.* The ModCP model is transformed and serialized.

UC III-2: *Load the model.* The software to be analyzed is the input and the reinstated ModCP model for that software is the output.

7.2.4 Time and space measurement

The use cases were implemented on ModCP using candidate serialization technologies. Each use case implementation for each serialization technology was run ten times, and the space and time

Listing 7.13: Warming up the serializer before recording the observations.

```
//cold startup
Serialize(prj);

Stopwatch reading = new Stopwatch();
Evaluate(prj);
...
```

was recorded for each run. `stopwatch.ElapsedMilliseconds` was used to capture the execution time of the run in milliseconds. We excluded the best and worst execution time from the ten runs and took the average of remaining eighth runs. For the space measurement, the space was same for the ten runs referring to each use case implementation. We used `FileInfo.Length` to retrieve the size of the serialized file in bytes.

The time involved to start up the serializer upon the first request is called the *cold startup time*.[1] Cold startup times are more expensive than the subsequent calls to the serializer [Chard et al., 2020]. Our experiment setup ensured that cold startup time was not the part of the recorded observation as shown in the Listing 7.13.

7.3 Results and analysis

We describe and analyze the results from our objective and subjective evaluation for the serialization technologies/libraries. The results are provided in terms of the measures used for each of the evaluations.

7.3.1 Objective evaluation

In this section we discuss the results of our study in terms of the use cases.

7.3.1.1 UC III-1: Store the model

Table 7.1 shows the time required to serialize the ModCP model. Protobuf-net took the longest time to translate the model object into a serialized object for all the datasets (with the possible exception of CDT for which BinaryFormatter crashed). BinaryFormatter performed efficiently in terms of time for the small datasets; however, as the size scaled up, the performance degraded and it failed to serialize for the largest dataset CDT because it exceeded the capacity of the serializer in terms of the number of objects (a maximum of 13 million objects are permitted to be serialized). Both the DataContract serializer and JSON.NET were more time-cost efficient than the other two, where DataContract performed slightly better than JSON.NET as the dataset size scaled up.

7.3.1.2 Space cost of the serialized object

Table 7.2 shows the space taken by the serialized ModCP model on disk. We found that Protobuf-net exhibited high space cost compared to the other three serialization technologies. BinaryFormatter and JSON.NET show comparable performance; JSON.NET had lower cost for smaller datasets, but BinaryFormatter scaled better and outperformed JSON.NET. The difference in the space performance between BinaryFormatter and JSON.NET may not be significant; however, the cost gap may increase for datasets larger than CDT. DataContract was the most inexpensive serialization technology in terms of space cost.

[1] Cold startup time. `https://docs.microsoft.com/en-us/archive/msdn-magazine/2008/march/clr-inside-out-improving-application-startup-performance`[2021/08/24], `https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/lambda-optimize-starttime.html`[2021/08/24]

Table 7.1: Serialization time (in milliseconds) for the candidate serializers on different datasets.

Serializer	JCommander	JSON	Ant-Ivy	Maven	JMeter	CDT
BinaryFormatter	290	597	1,857	5,012	14,232	fail
DataContract	**128**	**179**	**311**	**737**	**1,353**	**24,303**
NewtonSoft JSON.NET	297	374	746	1,179	4,588	32,319
Protobuf-net	539	525	3,949	9,078	16,623	318,198

7.3.1.3 UC III-2: Load the model

Table 7.3 displays the time involved during the deserializing of the encoded object (output of the serialization process) and reinstating the ModCP model. The performance of BinaryFormatter serializer was worst among the candidate serializers and we found that the deserialization time increase dramatically with the increase in the size of the dataset. Protobuf-net displayed an average cost in the comparative study. Again, DataContract serializer and JSON.NET were time-cost efficient than the other two, where DataContract performed slightly better than the JSON.NET as the dataset size was scaled up.

7.3.2 Subjective evaluation

In this section, we present the results of our subjective evaluations. We discuss in detail the information that we gathered and how the serialization technologies under consideration perform for specific measures.

7.3.2.1 Difficulty in implementation

As discussed in Chapter 6, SBS require careful inspection and intensive decoration of the classes and members to be serialized, as compared to FOGS where all the members are automatically

Table 7.2: Size of the serialized object (in MB) generated by the candidate serializers.

Serializer	JCommander	JSV	Ant-Ivy	Maven	JMeter	CDT
BinaryFormatter	6.475	10.44	33.44	78.81	187.32	fail
DataContract	**4.72**	**9.80**	**23.19**	**64.61**	**156.13**	**1,671.11**
NewtonSoft JSON.NET	5.32	11.64	29.15	78.44	192.38	1,939.33
Protobuf-net	31.93	34.78	154.8	388.01	901.3	3,339.85

Table 7.3: Deserialization time (in milliseconds) of the candidate serializers.

Serializer	JCommander	JSON	Ant-Ivy	Maven	JMeter	CDT
BinaryFormatter	435	998	6,514	38,369	261,275	fail
DataContract	**152**	**242**	**447**	**1,085**	**2,603**	**25,903**
NewtonSoft JSON.NET	208	290	498	1,146	2,817	27,411
Protobuf-net	505	548	2,539	8,684	12,831	282,546

serialized by the serializer. Another difficulty in SBS over FOGS is the requirement to pass the parent–child class relationship information to the serializer in order to execute the serialization process correctly. BinaryFormatter, being a FOGS, required minimal effort to implement in SDATs. In contrast, the other three considered technologies were SBS and involved higher efforts in terms of understanding the serialization technique library as well as understanding the structure of the considered ModCP. The data format of DataContract and NewtonSoft JSON.NET is human readable, whereas BinaryFormatter and Protobuf-net uses binary format which is not human legible. Although the human legible format of the serialized object does not offer any great insight, it was helpful to corroborate the decorated structure results during the implementation phase of the serialization technology.

7.3.2.2 Support

BinaryFormatter and DataContract are well documented being the official serialization technologies offered by C#. The documentation of Protobuf-net is available on GitHub, which is insufficient for industrial software developers. Support and additional documentation for Protobuf-net is requested on Stack Overflow, where the contributors including the author assist the community. NewtonSoft JSON.NET has proper documentation; however, it lacks user support.

7.4 Discussion

In this section we discuss remaining issues relative to this study.

7.4.1 Updating the model

Object serialization technologies do not support modification of the serialized object. In the event of a change in the code of the software being analyzed, the updated model from the memory of ModCP needs to be translated and serialized again. We suggest to run the serialization process in background in such events as a practical solution unless serialization technologies should evolve to include efficient modifiability.

7.4.2 Other object serialization technologies

We studied the capabilities of select C# object serialization technologies and narrowed them to finalize our candidate serialization technologies based on the four determining factors. While careful consideration was undertaken when exploring available serializers, we recognize that other serialization technologies could have been valid. Yamldotnet, an alternative to our candidate technologies, is a serialization library that allows the conversion of the object into a file conforming to the YAML format, a text serialization format similar to XML that is portable and platform-independent. Although Yamldotnet does not allow serialization of private members by default, the serializer can adjusted to allow it. Similarly, Apache Avro, a row-oriented data serialization technology present within Apache's Hadoop project, is another possible alternative; however, the documentation of Apache Avro for C# is almost absent and a user may benefit from prior knowledge of Java implementation (Apache Avro for Java has proper documentation).

7.4.3 Threats to validity

In this section we discuss the threats to validity of our work.

7.4.3.1 Internal Validity

Since we required a baseline for our results to compare for improvement, we selected the datasets used by Men [2018] to provide a clear picture of the change in performance measures. However, in doing so we may not always get a dataset representative of real world systems as the dataset includes software systems that may not accurately represent the models that may be generated for source code written in other programming languages.

7.4.3.2 External Validity

We evaluated the database technologies (in Study II), as well as the serialization technologies, on the same Windows machine since ModCP currently exists only for this machine, minimizing the background processes in all the cases. Due to the results being obtained for all the studies from the same OS under the same conditions, we manage consistency in the environment used to carry out our experiments so slight differences do not greatly impact the results as the relative differences in results are more significant in our evaluations.

7.4.3.3 Construct Validity

The threats related to subjective measures discussed in Section 4.5.4.3 also apply to this study. For the time measurement (objective measure), as discussed in Section 7.2.4, we eliminated the best and worst time measurements and took the average of remaining runs in order to reduce the variance and make the results more realistic.

7.5 Conclusions

As our results indicate (the combined time for serialization and deserialization is summarized in Table 7.4), DataContract clearly outperformed all the other candidate serialization technologies in storing and reloading the ModCP model. The space required on disk to store the model was also the lowest for DataContract (see Table 7.2).

BinaryFormatter required a slightly more space on disk to store the serialized model than did

Table 7.4: Combined time required (in milliseconds) to serialize and to deserialize the ModCP models, for various datasets.

Approach	JCommander	JSV	Ant-Ivy	Maven	JMeter	CDT
ModCP baseline	5,240	5,815	28,593	53,942	137,712	2,290,048
BinaryFormatter	725	1,595	8,371	43,381	275,507	fail
DataContract	**280**	**421**	**758**	**1,822**	**3,956**	**50,206**
NewtonSoft JSON.NET	505	664	1,244	2,325	7,405	59,730
Protobuf-net	1,044	1,073	6,488	17,762	29,454	600,744

DataContract. The time involved in storing the model was comparable to other candidate technologies; however, the deserialization time in loading the model was extremely high for larger datasets, thus exhibiting poor scalability of the technology and a poor option to reduce the re-computation time of SDATs. BinaryFormatter was unable to implement the use cases for the largest dataset, CDT, due to its limit on serializing no more than 13 million objects. [2]

As for Protobuf-net, the space required on disk was the highest compared to other technologies. It also took the most processing time to store the model. However, we observed that it scales better than BinaryFormatter. The loading of the model was more costly than DataContract and NewtonSoft JSON.NET.

Although for NewtonSoft JSON.NET the time involved in storing the model was approximately double that of DataContract for the smaller datasets, it scaled well and demonstrated a promise of better performance for even larger datasets than CDT. The performance of NewtonSoft JSON.NET was highly comparable to the DataContract serializer in loading the model back in ModCP, being only slightly more costly than DataContract. Similarly, the space taken by JSON.NET was the second lowest of all technologies considered after DataContract.

The subjective analysis of the object serialization technologies indicates that BinaryFormatter exhibits the least difficulties during implementation, with good support and documentation. NewtonSoft JSON.NET involves greater difficulty in implementation and does not have great support; however, it has good documentation. Protobuf-net required the greatest efforts during implementation (being an SBS), its documentation is not comprehensive, and it has limited support on open-source community forums. DataContract required high effort during implementation; however, it is an official .NET serializer which comes with good documentation and support.

Based on our study, we conclude that the use of object serialization, especially the DataContract serializer, offers reduced re-computation cost for ModCP. Potential, further reduction through optimization will be the subject of the subsequent chapter.

[2] Binaryformatter issue related to maximum array size. `https://github.com/dotnet/runtime/issues/20569`[2021/08/24]

7.6 Summary

We conducted a comparative study of the object serialization technologies and described the measures we used to evaluate them in order to analyze any potential for using them in SDATs. We presented and discussed the setup for evaluation and our results. We found that DataContract serialization technology is the most inexpensive in terms of the time required to execute the processes to store and retrieve the model and requires the least space on disk to store the translated models. Object serialization technologies are promising options to reduce the re-computation time involved in the SDATs.

Chapter 8

Study IV: Optimizing DataContract Serializer in ModCP

In this chapter, we explore how to serialize the ModCP model again via the DataContract serialization technology from Chapter 7, but seeking optimizations to reduce the re-computation time further. The goal of this study is to address the following research question:

> **RQ4**: *How can we optimize the DataContract object serialization implementation to reduce the rebuild time further?*

We find that breaking the model into submodels, to serialize them individually, increases the size of the serialized object and increases the time involved in translating and reinstating the model. Thus, we were unable to further optimize the results from Study III.

We review some information about the ModCP model which will have bearing on this study, in Section 8.1. We use the same objective performance measures discussed in Section 7.1. The same datasets from Studies II and III were used in this study, which were discussed in Section 4.2. The study setup is explained in Section 8.3. We present the results and our analysis thereof in Section 8.4. Remaining points for discussion, relative to this chapter, are given in Section 8.5. Section 8.6 presents our conclusions in response to the research question.

8.1 Interdependencies between the submodels of ModCP

To understand the nature of further potential optimizations that we explore in this chapter, we must revisit details of the ModCP model to understand the interdependencies between its submodels.

ModCP utilizes both a syntactic model and semantic model of the source code it must analyze. The syntactic model consists of the set of abstract syntax trees (ASTs) that are constructed as the source code is parsed. The semantic model consists of the extended call graph (ECG), class hierarchy graph (CHG), and a set of modularized program dependence graphs (mPDGs). All parts of the semantic model make reference to nodes in the syntactic model, specifically to type declarations (for ECG, CHG, and mPDGs), method declarations (for ECGs and mPDGs), and field declarations (for mPDGs).

Given a model to serialize, a good serializer will take advantage of references to already serialized objects in order to avoid their redundant re-serialization, which would otherwise waster both space and time.

The study we describe in this chapter examines the interplay between partial serialization of the ModCP model and the need for re-serialization of some elements of the syntactic model.

8.2 Performance measures

We used the processing time of the use cases and the space taken by the object serialization technique as the objective measures in this study. (The subjective measures of the candidate object serialization techniques were already evaluated in Chapter 7, so we do not consider them further.)

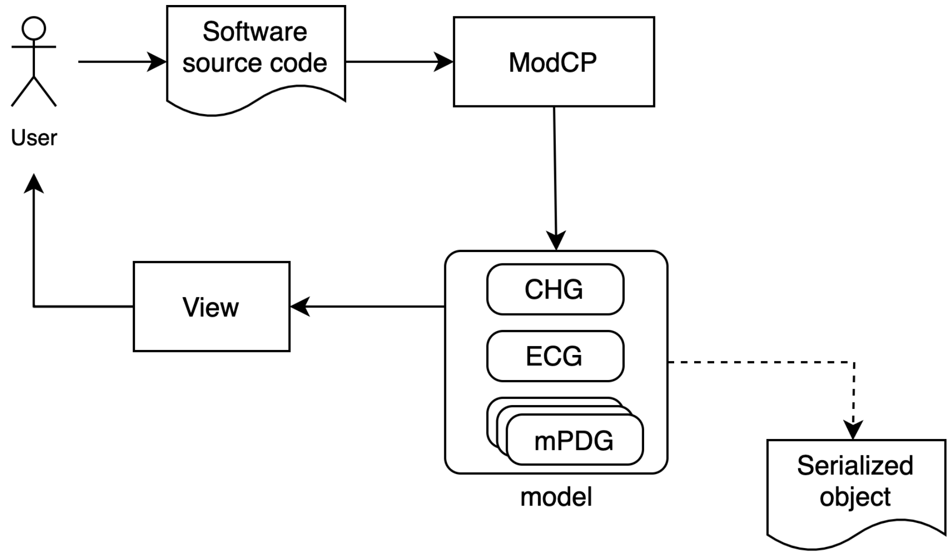


Figure 8.1: Graph store operation process.

8.3 Setup for evaluation

8.3.1 Procedure

We studied the ModCP model in depth to understand the processing times involved in building the model. As discussed earlier, the ModCP (semantic) model comprises three kinds of submodel: a class hierarchy graph, an enhanced call graph, and modularized program dependence graphs (mPDGs). As shown in Figure 8.1, once the source code of the dataset is provided as input to ModCP, the source code is first parsed to a syntactic model. The semantic model is thereafter constructed from an analysis of the syntactic model. The building of both the syntactic and the semantic models depends upon the size of the dataset. For instance, the size of the class hierarchy graph will depend upon the number of classes in the source code of the datasets. Similarly, the size of the semantic model depends upon the number of methods present in the dataset source code. Furthermore, the size of the submodels and the complexity of the model depends upon the declarations and references of entities in the source code. We gathered these specifics of the model and studied this information to seek optimizations for the rebuild approach, in order to improve upon the results from Study III from the DataContract serializer.

8.3.2 ModCP model specifics

The ModCP model is different for every dataset, and its size and complexity depend upon the nature of the dataset to be analyzed. We gathered information from each model to study the specifics and their impact on model build time by ModCP.

Table 8.1 shows the values of metrics for basic dataset properties and for properties of the ModCP model, relative to each dataset. The properties include (counts of) statements, classes, fields, and functions, each providing the number of these entities in the dataset. The row "# Class relationships" gives the number of relationships between the classes within the dataset, and "# Function relationships" provides the number of unique function calls across all functions. The row "# mPDGs" provides the number of mPDGs built for each dataset in the model, and "Model build time" gives the time it takes to build the model, in milliseconds. The datasets in the columns are arranged in increasing order of their model build times, JCommander being the one requiring the least time to build its models and CDT being the one taking the most time. The increase is relative to the complexity of the code as shown by "# Class relationships" and "# Function relationships."

8.3.3 ModCP model processing time (Baseline)

We analyzed the datasets in ModCP and captured the processing times of different stages during the building of a model. These processing times are described in Table 8.2 and are used as baseline in our experiment. Building the syntactic model involves parsing the input source code and generating

Table 8.1: Inherent dataset metrics and ModCP model metrics.

Dataset	JCommander	JSV	Ant-Ivy	Maven	JMeter	CDT
# Statements	1,747	2,689	9,789	23,226	59,632	592,710
# Classes	54	156	150	546	1,065	14,366
# Functions	525	863	2,411	6,229	16,245	150,392
# Fields	61	223	432	971	4,488	36,182
# Class relationships	27	84	66	191	768	8,783
# Function relationships	604	822	3,217	7,234	22,951	229,405
# mPDGs	322	606	1,488	4,100	9,564	90,332
# Nodes in mPDGs	4,442	7,845	24,204	61,215	148,892	1,560,112
# Edges in mPDGs	6,888	12,678	38,018	99,286	235,586	2,463,790
Model build time (ms)	5,240	5,815	28,593	53,942	137,712	2,290,048

the ASTs to represent it. Building the semantic model involves analyzing the syntactic model and generating the enhanced call graphs, class hierarchy graphs, and mPDGs (one for every method in the input source code). The row “Semantic model” refers to the time taken in this described activity (this is sometimes slightly larger than the sum of the build times for the call graph, call hierarchy, and mPDGs because of overhead arising from the timing infrastructure). The rows “Call graph”, “Class hierarchy”, and “mPDGs” refer to the time required by ModCP to build the call graph, class hierarchy, and mPDGs, respectively.

Based on these baseline results, we find that building the syntactic model is very expensive compared to building the semantic model. When observing the semantic model even deeper, we find that mPDGs (sub-sub model) are most expensive compared to call graph (sub-sub model) and class hierarchy graph (sub-sub model). Based on these details, we propose two ideas for potential optimization:

1. serializing the syntactic model and building the semantic model traditionally might decrease the rebuild costs even further;
2. since the mPDGs were the most costly parts of the models to build, we suspected that serializing/deserializing them while rebuilding the remainder might result in overall faster performance.

8.3.4 Different serialization approaches

In Study III, our approach was to (de)serialize the full model of ModCP, which allowed the DataContract serializer to thoroughly avoid redundant serialization of syntactic model elements. Based

Table 8.2: Detailed information on the build times of the submodels of ModCP, in milliseconds.

Submodel	JCommander	JSV	Ant-Ivy	Maven	JMeter	CDT
Syntactic model	4,995	5,535	27,594	51,272	129,404	2,124,923
Call graph	36	33	95	240	1,220	63,575
Class hierarchy	7	1	3	7	5	765
mPDGs	199	245	1,003	2,422	7,083	100,726
Semantic model (total)	245	280	1,099	2,670	8,308	165,125

on the detailed information on build times and our ideas for potential optimizations, we took three different approaches to improve the rebuild time of ModCP using the DataContract serializer. These approaches are all used to optimize the costly stage in the model (re)build and are discussed below.

1. **Approach 1: Serialize the submodels individually.** In this approach, we first serialize the syntactic model and semantic model individually, meaning that the serializer would have to re-serialize any elements referenced by the semantic model from the syntactic model. Later, when the model is to be rebuilt, we deserialize the semantic model and syntactic model, and merge them together to reinstate the overall model. The overall model is then used to serve the software analysis for the developer.

2. **Approach 2: Serialize the syntactic model and rebuild the semantic model.** In this approach, we only serialize the syntactic model to the datastore. In the rebuild process, the syntactic model is deserialized and conventional methods from ModCP are used to build the semantic model.

3. **Approach 3: Serialize the syntactic model and serialize each mPDG individually.** Similar to Approach 1, we serialized the syntactic model, but we serialized each mPDG individually as for the semantic model. The idea behind this approach is to cut the time of rebuilding the model significantly by only deserializing the mPDGs which are needed when analysis is requested.

8.3.5 Use cases

Similar to Study III, we have two use cases for this study.

UC IV-1: *Store the model.* The ModCP model, or only some of its submodels, is serialized.

UC IV-2: *Load the model.* The software to be analyzed is the input and the reinstated ModCP model for that software is the output.

8.3.6 Time and space measurement

Similar to Study III, the use cases were implemented with the described approaches on ModCP using DataContract serialization technology. Each combination of use case and approach was run ten times, and the space and time was recorded for each run. We excluded the best and worst execution time from the ten runs and took the average of remaining eight runs. We followed the same space measurement described in Section 7.2.4.

8.4 Results and analysis

In this section, we present the findings for the approaches that we discussed earlier.

8.4.1 Approach 1 evaluation

We serialized the submodels of ModCP, stored them to disk, and deserialized them again to simulate the rebuild process of ModCP using the DataContract serializer. Table 8.3 displays the processing time in milliseconds to serialize the submodels, where “syntactic” is the serialization time of the syntactic model and “semantic” is the serialization time of the semantic model. Table 8.5 displays the processing time in milliseconds to deserialize the submodels where “syntactic” is the deserialization time of the syntactic model and “semantic” is the deserialization time of the semantic model. Table 8.4 shows the space required to serialize the submodels to disk, where “syntactic” is the space required by the syntactic model and “semantic” is the space required by the semantic model.

8.4.2 Approach 2 evaluation

We serialized/deserialized only the syntactic model and used it to build the semantic model in the conventional way to simulate the rebuild process. Table 8.6 shows the time involved in serializing only the syntactic model. Table 8.7 shows the processing times, in milliseconds, involved in various steps to rebuild the complete ModCP model; “syntactic” is time taken to deserialize and

Table 8.3: Approach 1, UC IV-1: The time taken to serialize the submodels individually, in milliseconds.

Dataset	Syntactic	Semantic
JCommander	67	52
JSV	125	105
Ant-Ivy	412	370
Maven	1,004	993
JMeter	2,470	3,824
CDT	38,925	65,828

Table 8.4: Approach 1, UC IV-2: The time taken to deserialize the submodels and reconstruct the full model, in milliseconds.

Dataset	Syntactic	Semantic
JCommander	95	80
JSV	180	156
Ant-Ivy	551	521
Maven	1,370	1,451
JMeter	3,531	3,471
CDT	40,284	42,645

Table 8.5: Approach 1: The space on disk required to save the submodels, in bytes.

Dataset	Syntactic	Semantic
JCommander	3,077,168	5,118,535
JSV	6,888,858	8,772,699
Ant-Ivy	14,293,452	27,429,096
Maven	45,685,117	71,803,812
JMeter	112,927,229	178,171,629
CDT	1,332,713,454	1,958,576,142

Table 8.6: Approach 2, UC IV-1: The time taken to serialize the syntactic model, in milliseconds.

Dataset	Syntactic
JCommander	67
JSV	125
Ant-Ivy	412
Maven	1,004
JMeter	2,470
CDT	38,925

reinstate the syntactic model, “semantic” refers to the time taken to rebuild the semantic model, and “overall” is the total time to reinstate the complete ModCP model (the sum of the other two). Table 8.8 shows the space taken by the serialized syntactic model on disk.

Table 8.7: Approach 2, UC IV-2: The time taken to deserialize the syntactic model and rebuild the semantic model, in milliseconds.

Dataset	Syntactic	Semantic	Overall
JCommander	95	245	340
JSV	180	280	460
Ant-Ivy	551	1,099	1,650
Maven	1,370	2,670	4,040
JMeter	3,531	8,308	11,839
CDT	40,284	165,125	205,409

Table 8.8: Approach 2: The space required to store the syntactic model, in bytes.

Dataset	Syntactic
JCommander	3,077,168
JSV	6,888,858
Ant-Ivy	14,293,452
Maven	45,685,117
JMeter	112,927,229
CDT	1,332,713,454

8.4.3 Approach 3 evaluation

Similar to Approach 2, we serialize the syntactic model using the DataContract serializer; however, we also serialize the individual mPDGs for each method. Table 8.9 shows the time required to serialize these elements; “syntactic” refers to the time taken to serialize the syntactic model, “mPDGs” refers to the time taken to serialize all the mPDGs, and “overall” refers to the total time involved in the process of serializing the submodels. Table 8.10 shows the time required to deserialize these elements and to rebuild the remainder of the complete ModCP model; “syntactic” refers to the time taken to deserialize the syntactic model, “semantic” refers to the time taken to deserialize all the mPDGs and to rebuild the ECG and CHG, and “overall” refers to the total time involved in the process of deserializing the submodels and rebuilding the complete ModCP model. Table 8.11 shows the storage space required; “syntactic” represents the space required for the syntactic model, “mPDGs” represents the space required for all the mPDGs, and “overall” represents the complete storage requirements on disk for this approach.

Table 8.9: Approach 3, UC IV-1: The time taken for serialization, in milliseconds.

Dataset	Syntactic	mPDGs	Overall
JCommander	67	41,939	42,006
JSV	125	71,453	71,578
Ant-Ivy	412	119,769	120,181
Maven	1,004	368,128	369,132
JMeter	2,470	1,000,764	1,003,234
CDT	38,925	3,633,481	3,672,406

Table 8.10: Approach 3, UC IV-2: The time taken to deserialize the submodels and rebuild the complete model, in milliseconds.

Dataset	Syntactic	Semantic	Overall
JCommander	95	36,971	37,066
JSV	180	57,231	57,411
Ant-Ivy	551	135,531	136,082
Maven	1,370	288,351	289,721
JMeter	3,531	845,889	849,420
CDT	40,284	2,922,894	2,963,178

Table 8.11: Approach 3: The space required to store the submodels, in bytes.

Dataset	Syntactic	mPDGs	Overall
JCommander	3,077,168	334,385,439	337,462,607
JSV	6,888,858	3,415,558,239	3,422,447,097
Ant-Ivy	14,293,452	5,236,358,121	5,250,651,573
Maven	45,685,117	18,648,328,769	18,694,013,886
JMeter	112,927,229	49,007,680,081	49,120,607,310
CDT	1,332,713,454	182,418,038,936	183,750,752,390

8.5 Discussion

In this section, we discuss remaining issues relative to this study.

8.5.1 Threats to validity

The same threats discussed in Section 7.4.3 are applicable to this study.

8.6 Conclusions

Table 8.12 compares the time required to rebuild the model for the five approaches. Similarly, Table 8.13 compares the time required to serialize the model (or submodels). Table 8.14 provides the space cost comparison between the five approaches to rebuild the model. Restoring the model in Approach 1 was promising for the smallest two datasets; however, the difference between the Study III results and those of Approach 1 increased as we scaled up to the larger datasets. Similarly, the time needed to store the model was lower for Approach 1 than for the Study III results, for the smallest dataset, but was significantly higher for the larger ones. Another observation was that

Table 8.12: The time required to complete the rebuild, or deserialization plus rebuild, for the five approaches, in milliseconds.

Dataset	Baseline	Study III	Approach 1	Approach 2	Approach 3
JCommander	5,240	152	175	340	37,066
JSV	5,815	242	336	460	57,411
Ant-Ivy	28,593	447	1,072	1,650	136,082
Maven	53,942	1,085	2,821	4,040	289,721
JMeter	137,712	2,603	7,002	11,839	849,420
CDT	2,290,048	25,903	82,929	205,409	2,963,178

Table 8.13: The time required to serialize the models (or submodels), in milliseconds.

Dataset	Baseline	Study III	Approach 1	Approach 2	Approach 3
JCommander	0	128	119	67	42,006
JSV	0	179	230	125	71,578
Ant-Ivy	0	311	782	412	120,181
Maven	0	737	1,997	1,004	369,132
JMeter	0	1,353	6,294	2,470	1,003,234
CDT	0	24,303	104,753	104,753	3,633,481

Table 8.14: The space required to store the models (or submodels), in bytes.

Dataset	Baseline	Study III	Approach 1	Approach 2	Approach 3
JCommander	0.0	4,714,968	5,118,535	3,077,168	340,462,607
JSV	0.0	9,844,353	8,772,699	6,888,858	3,427,447,097
Ant-Ivy	0.0	23,193,871	27,429,096	14,293,452	5,270,651,573
Maven	0.0	64,611,257	71,803,812	45,685,117	18,734,013,886
JMeter	0.0	156,134,119	178,171,629	112,927,229	49,220,607,310
CDT	0.0	1,671,108,717	1,958,576,142	1,332,713,454	184,750,752,390

Approach 1 performed worse than Approach 2 in UC IV-1, but better in UC IV-2. Approach 2 involved higher space costs than Study III results and Approach 2.

Restoring the model in Approach 2 was expensive compared to Study III and to Approach 1 for the smallest dataset and performed even worse for the larger datasets. Storing the model in Approach 2 was the cheapest for the smaller datasets, JCommander and JSV. However, it did not scale as well as Study III and took almost double the time for the largest dataset. Approach 2 exhibited the lowest space cost (not counting Study III) because it involved saving the syntactic model only.

Storing and loading the model in Approach 3 demonstrated terrible performance and was even more expensive than the baseline. The space cost for Approach 3 was also the greatest of all, with a large difference compared to the other approaches.

Since the modification of serialized object files is not supported, our idea was to break the models into smaller models so that updating the datastore (serialized submodel files) with the modified ModCP model would only require serializing fewer and smaller submodels (resulting in additional savings for big datasets). We recognize two sources of overhead from our optimization approaches:

1. An expensive, yet necessary, element that increased the cost of serializing the submodel was serializing a few redundant fields to make the translated serialized files mergeable during the rebuilding process of the ModCP model.

2. Serialized submodels also required more function calls, including reading/writing of several files.

Approach 1 involved breaking down the model in two submodels (syntactic and semantic); thus, the impact of the overheads is the least. Approach 3 involved breaking down the model into the syntactic model, the class hierarchy graph model, the call graph model, and a huge number of mPDGs, one per method in the software being analyzed. As the results indicate, the additional cost due to these overheads was dramatic and thus unfeasible. Approach 2 was a hybrid approach that involved a derivation of costs from both ModCP and the DataContract serializer. We found that the cost of using Approach 2 was higher than Approach 1.

Although Approaches 1 and 2 were more expensive than the results from Study III, we found that rebuilding the ModCP model was still much cheaper than the traditional approach (baseline). Even though our three optimization approaches did not yield improved results for ModCP compared to the Study III results, we believe that they still have potential and may demonstrate reduced rebuild times for some SDATs which possess invariant submodels after modification of the software being analyzed. Depending upon the nature of the model and its submodels, there is scope of reduction in the processing time involved in the (de)serialization process—but unfortunately not for ModCP as it stands.

8.7 Summary

We performed a case study to reduce the re-computation time further by experimenting with certain approaches where we divided the model into submodels. We discussed our approach, the setup of the study, and our results. We found that storing and loading of the submodels and re-creating the main model comes with extra costs that negate the improvements achieved through Study III.

Chapter 9

Discussion

In this chapter, we discuss the choices we made in the four studies, and their impacts on the results.

9.1 Deliberation of studies

In Study I, we explored relational and non-relational databases to simulate the mechanism of storing an SDAT model. We artificially generated scale-free graphs that mimic the dependency graphs of ModCP. Although there exist several scale-free graph generation models, we followed the Barabási—Albert model for its ability to generate real-world-sized graphs. Our industrial partner, Find it EZ Software Corporation, is migrating from relational databases to non-relational databases thus we emphasized specifically comparing these two types of technologies in this study. We also considered and included modifying the CSV datasets in their original format using Python programming language because of its ability and potential to be used a datastore for SDATs. Since the Barabási-–Albert model has been implemented using the networkx Python library, we continued the usage of Python programming language to handle the CSV datasets despite the availability and knowledge of other programming languages. We implemented four operations—create, retrieve, update and delete—using eight queries. We ran these eight queries ten times on each candidate database technology and averaged the findings. Our results indicated that using Python-CSV as a database technology could be viable for small-sized datasets (or SDAT model), but the performance quickly degrades substantially as the size of the dataset increases, and would thus be impractical for our needs. Our results concluded PostgreSQL will serve best the SDATs with high demand to basically only store and retrieve the new models, whereas Neo4j would be more suitable for SDATs where maintaining the existing models in the datastore is of priority, being done frequently.

Therefore, we implemented connectors to both Neo4j and PostgreSQL within ModCP, to understand the feasibility of these two database technology in reducing the rebuilding time of previously analyzed software. Our use cases were: storing the model, loading the model back, and modifying the model. For both technologies, a connection was established and the query was sent and processed by the database technology. We used the same datasets used in the evaluation of the prototype ModCP tool in order to ensure a fair comparison of ModCP with database support against traditional ModCP. The results of our study indicated that integration of database technology was far from feasible because of the extremely high costs involved in storing, maintaining, and loading the model of ModCP. We found that the overhead of using a connector results in unacceptable performance and thus we discarded the possibility of using database technologies in ModCP.

We then explored object serialization, conversion of in-memory object representation into stream of bytes, and its potential usage in ModCP to address the issue of long traditional rebuild time. We studied some of the object serialization technologies available and found that many of them would not work in our context. We used four determining factors to select our candidate serialization technologies: C# language support, ability to serialize private class members, capturing of references, and graph support (these factors may change for other SDATs). Based on these factors, we selected the BinaryFormatter, DataContract serializer, NewtonSoft Json.NET and Protobuf-net. There were other serialization technologies satisfying our requirements as well; however, they lacked sufficient support and documentation. We serialized the ModCP model using these four technologies and loaded the model back to simulate the rebuilding event. These two use cases were run ten times each, the best and worst run times were excluded, and the average was taken of the remaining eight observations. This ensured the elimination of the extreme high and low observations. Our findings

Table 9.1: The rebuild cost reduction achieved in ModCP using DataContract object serialization technology.

	JCommander	JSV	Ant-Ivy	Maven	JMeter	CDT
Baseline	5,240	5,815	28,593	53,942	137,712	2,290,048
DataContract	280	421	758	1822	3956	50206
Reduction	18.71428571	13.81235	37.72164	29.60593	34.81092	45.61303

indicated that BinaryFormatter exhibits high deserialization cost compared to other candidate technologies as the dataset size increased. Protobuf-net demonstrated highest serialization cost as well as highest space cost of storing the model data on disk. The time involved in deserialization using DataContract and NewtonSoft Json.NET was comparable, with DataContract performing slightly better. However, DataContract clearly outperformed Json.NET in the serialization process (storing the ModCP model) and thus it is the most feasible option among the candidate serialization technologies.We found that the performance improvement from the DataContract serializer increased with the size of the datasets. Table 9.1 shows the comparison of rebuilding the ModCP model using DataContract serialization technology and the traditional approach. We observed that the model rebuild cost was reduced at least 13 times for the smallest dataset (JCommander) and 46 times for the largest dataset (CDT). The implementation of DataContract serialization not only offered faster rebuilds than the traditional rebuilding of ModCP, but also displayed good scalability. Figure 9.1 shows how the performance scales linearly relative to the size of the systems being analyzed.

Object serialization technologies do not offer modification of the serialized object; therefore, in the event of even minor changes in the software being analyzed, the model needs to be completely serialized again. In order to minimize the time elapsed in multiple and frequent serialization of ModCP model, we decided to serialize the submodels instead; this would lead to only serializing a part of model (submodel) rather than serializing the whole model again. We identified three optimization approaches for implementing the DataContract serializer in ModCP; Approach 1 was to serialize the syntactic and semantic models individually, Approach 2 only required serializing the syntactic model with the semantic model being re-built using the traditional approach, and Approach 3 involved serializing the syntactic model and serializing each mPDG individually. We

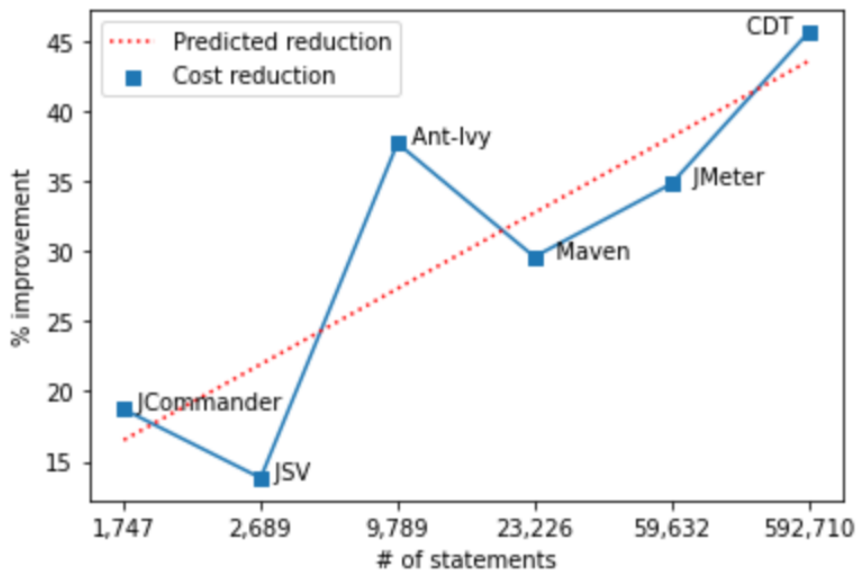


Figure 9.1: Scalability of cost reduction for ModCP in using DataContract.

implemented the same use cases and analyzed the same datasets in ModCP in this study. Our results indicated that the cost of rebuilding using the three optimization approaches was higher than the Study III complete model serialization. We identified and recognized two sources of high overhead: (1) serialization of redundant fields since they cannot be referenced, and (2) reading/writing several files (90,000+ for largest dataset, CDT) including a high number of serializer calls.

As our analysis shows, building the syntactic model and (de)serialization of it is the most expensive aspect (except the Approach 3). Currently, ModCP parses all the Java class files when building the ModCP model the first time (type declaration). Once the model is built, and any modifications are introduced in the software being analyzed, the parser parses only the file in which the change is made and updates its concerned type declaration. However, if the parsing process of ModCP is conducted for even smaller granularity level, such as method declaration, then the parsing time would be reduced and the repetitive (de)serialization process of such a small submodel would likely have minimal impact on the cost of updating the serialization datastore. Confirming this hypothesis will require further research.

9.2 Impact of research

The results of our research are relevant to any new SDATs being developed or any existing SDATs that may require offline storage. Our results would be especially instrumental in cases where decisions need to be made based on factors that may be similar to the criteria for which we analyzed the storage mechanisms. The considered criteria broadly covers the use cases which are generally meaningful for analyzing storage mechanisms. We have provided the results based on use cases and do not conclude any single technology to be a universally superior solution as there cannot be one that will unequivocally meet the requirements of varying SDATs in which the trade-off between time and space differs significantly. We hope that this work provides useful insight into how the storage mechanisms perform for different criteria.

9.3 Future work

Our optimization approaches (in Study IV) did not yield the desired results. Currently, object serialization technologies do not offer modification of the serialized object, and thus maintaining the up-to-date ModCP model may become expensive where frequent modification of the software being analyzed is done. As discussed earlier in Chapter 9, in order to make object serialization technology more suitable for ModCP, the parsing operation should be performed at a finer granularity (perhaps method-level). In the future, we would like to take a new route about the way we (de)serialize the syntactic model, such as serializing the type declaration information without its member details, serializing the method and field declarations separately, and serializing the mPDGs separately. Furthermore, replacing the ANTLR parser generator (as currently employed in ModCP) with JavaCC may reduce parsing speed significantly at the cost of developing slightly more complex grammars; other exploratory work in our laboratory suggests that this is a viable route for future exploration.

In any case, determining how to modify the submodels in order to be able to serialize only the modified parts is a potentially important roue for future investigation.

Lastly, we look forward to investigating other concrete SDATs to gain insight into how the datastore integration would impact the cost of recomputation of different models. We hope that further exploration in this area would allow us to generalize the results.

Chapter 10

Conclusion

Software systems require constant maintenance in an ever-changing business model or they risk becoming outdated. Large software systems make it challenging to add or remove functionality; therefore, developers often rely on software development and analysis tools (SDATs) to help in easing this process.

SDATs requires building of analysis models for the software being analyzed by the developers. The time involved in building these analysis models can involve considerable high wait times for the developers before they can actually begin analyzing the software. The inconvenience due to this interim time gets worse when analyzing multiple software projects at the same time as the process of building has to be repeated since it is stored in volatile memory. In our research, we considered and addressed this issue in ModCP, a change propagation tool. The loss of analysis model can be mitigated by incorporating a storage support, where the model can be stored and fetched back to ModCP, avoiding the re-computation using the traditional approach.

Existing work in this area ranges from providing comparisons of various relational and non-relational databases, and serialization techniques. The recurring issue in these works revolves around the lack of a comprehensive study that encapsulates the performance on these storage mechanisms in real world systems. The size of the system under study matters, as does the complexity of the source code; many of the previous works rely on simple objects or toy code-like projects to evaluate performance. However, this is problematic due to the nature of storage mechanisms;

the rebuilding of a model for larger and more complex systems requires more space, and takes more time than smaller and simpler systems. Another shortcoming is the sole focus on quantitative aspects, and not addressing many other subjective factors that would play a critical role when datastore technology is integrated with an SDAT. Lastly, the additional cost of implementing datastore technology in a software, a critical factor in our work, has not been addressed.

In our attempt at studying the potential usage of storage mechanisms in SDATs to reduce the interim wait time during re-build of the same underlying software, we tried to answer different research questions regarding the mechanisms under evaluation:

RQ1: *How do different database technologies perform in comparison to one another in executing store, delete, and change operations over analysis graphs in a real-world software system?*

We explored relational and non-relational databases, studied their operation mechanisms and considered their benefits to use as a potential storage mechanism in ModCP. We selected Python-CSV, MySQL, Neo4j and PostgreSQL as our candidate data-store technologies and conducted Study I to measure and compare the performance of each of them. Our results indicated that for the identified use-cases in Study 1, Python-CSV lack scalability as the data becomes more complex and grows in size. MySQL display great performance storing the smaller graphs, but didn't scale well for larger datasets. Neo4j and PostgreSQL were selected as two viable options due to each of these performing well in two operations: *delete* and *change* operations for Neo4j, *store* and *retrieve* operations for PostgreSQL.

RQ2: *What is the impact of adding/integrating datastore technology support to an SDAT on the re-computation cost of the analysis model?*

Based on the results of previous study, we integrated and implemented Neo4J and PostgreSQL in ModCP to identify the impact of the database technology support on re-computation cost. We found that storing the model was cheaper for PostgreSQL while retrieving the model back in ModCP was cheaper for Neo4j. Updating and deleting the model had no significant difference between the two. We deduced that the impact of integrating a database technology introduces a significantly high overhead as the time to execute queries far exceeds the traditional approach, therefore, this makes both PostgreSQL and Neo4j unrealistic for us to use.

Based on the infeasible results from Study 2, we explored another datastore technology known as object serialization. We described and discussed the types of serializers and the formats they translate the model object into. We considered several object serialization technologies for C#, however, we narrowed our list to four candidate technologies based on the ability to serialize private class members and support graph and references during serialization.

Thereafter, we performed a comparative study between (BinaryFormatter, DataContract, NewtonSoft JSON.NET, and Protobuf-net) to answer the following research question.

RQ3: *How do object serialization technologies compare in terms of space and time cost required for model data storage and access mechanism when integrated with ModCP?*

We studied and selected four candidate object serialization technologies based on four determining factors. We implemented the candidate serialization technologies, analyzed the six previously used datasets from Study-2 and stored/retrieved the ModCP model to simulate recomputation process. We found that BinaryFormatter exhibit high deserialization cost and Protobuf-net involved high space and time costs. NewtonSoft Json.NET and DataContract had comparable deserialization times, however DataContract outperformed all the other candidate serialization technologies and was the most inexpensive option to store and retrieve the model; it required the lowest time to implement the use cases and least space on disk to store the translated models.

We found that DataContract serialization technology is the most inexpensive in terms of the time required to execute the processes to store and retrieve the model and requires the least space on disk to store the translated models. Object serialization technologies are promising options to reduce the re-computation time involved in the SDATs.

RQ4: *How can we optimize the DataContract object serialization implementation to reduce the rebuild time further?*

We analyzed the models generated by the ModCP for the same six datasets we used in Study 2 and Study-3. We proposed three optimization approaches to serialize the ModCP model using variations of submodels. We found that storing and loading of the ModCP (sub)model using our proposed approach did not yield reduced recomputation cost compared to the previously achieved reduction in Study-3. The high overhead of storing the redundant data, the inability of referencing fields across other submodel and high number of read/write operations made these approaches unfeasible.

Our research addresses the undesirable downtime in the developer's workflow due to high cost involved in recomputing complex models of SDATs for the software that had been analyzed before. We explore the potential benefits of implementing datastore technologies in ModCP, an industrial change propagation tool. Related work in this area lacks real-world sized datasets, consideration of quantitative aspects in the analysis, and restricts to storing/loading limited primitive data type fields. Furthermore, the majority of existing studies mainly focus on comparing the datastore technologies alone, whereas, our work considers the impact of integration of database technologies in an SDAT (or a software in general) and thereafter comparing the performance allowing us to study the additional costs introduced. The results of our research are relevant to other existing or new SDATs, providing a detailed understanding of the impact of integrating datastore technology on the re-computation cost. The limitation of serialization technologies to modify the stored/serialized model indicates the need of further investigation. Lastly, in due course, future work would concern a deeper dive into the evaluation of other SDATs to generalize the results of datastore integration.

Bibliography

Aayush Agarwal, Subhash Gupta, and Tanupriya Choudhury. Continuous and integrated software development using DevOps. In *Proceedings of the International Conference on Advances in Computing and Communication Engineering*, pages 290–293. IEEE, 2018.

Apostolos Ampatzoglou, Stamatia Bibi, Paris Avgeriou, Marijn Verbeek, and Alexander Chatzigeorgiou. Identifying, categorizing and mitigating threats to validity in software engineering secondary studies. *Information and Software Technology*, 106:201–230, 2019. doi: 10.1016/j.infsof.2018.10.006.

Bente C. D. Anda, Dag I. K. Sjøberg, and Audris Mockus. Variability and reproducibility in software engineering: A study of four companies that developed the same system. *IEEE Transactions on Software Engineering*, 35(3): 407–429, 2008. doi: 10.1109/TSE.2008.89.

Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. Graph-based malware detection using dynamic analysis. *Journal in Computer Virology*, 7(4):247–258, 2011. doi: 10.1007/s11416-011-0152-x.

Hugo Andrade, Federico Giaimo, Christian Berger, and Ivica Crnkovic. Systematic evaluation of three data marshalling approaches for distributed software systems. In *Proceedings of the Workshop on Domain-Specific Modeling*, pages 71–76, 2015. doi: 10.1145/2846696.2846705.

Renzo Angles. A comparison of current graph database models. In *Workshop Proceedings of the IEEE International Conference on Data Engineering*, pages 171–177, 2012. doi: 10.1109/ICDEW.2012.31.

Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39, 2008. doi: 10.1145/1322432.1322433.

Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. doi: 10.1126/science.286.5439.509.

Shalini Batra and Charu Tyagi. Comparative analysis of relational and graph databases. *International Journal of Soft Computing and Engineering*, 2(2):509–512, 2012. doi: 10.1109/CAIS.2018.8441982.

Oren Ben-Kiki, Clark Evans, and Ingy döt Net. YAML ain't markup language (YAML) version 1.2. Working Draft 2008-05-11, 2008. URL `https://yaml.org/spec/spec.html`.

Paul E. Black. *Dictionary of algorithms and data structures*. National Institute of Standards and Technology, 1998. URL `https://xlinux.nist.gov/dads/`.

Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML). *World Wide Web Journal*, 2(4):27–66, 1997.

Pierce G. Buckley and Deryk Osthus. Popularity based random graph models leading to a scale-free degree sequence. *Discrete Mathematics*, 282(1-3):53–68, 2004. doi: 10.1016/j.disc.2003.10.020.

Bobby J. Calder, Lynn W. Phillips, and Alice M. Tybout. The concept of external validity. *Journal of Consumer Research*, 9(3):240–244, 12 1982. doi: 10.1086/208920.

Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. FuncX: A federated function serving fabric for science. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing*, pages 65–76, 2020.

Edgar F. Codd. Relational database: A practical foundation for productivity. In *Readings in Artificial Intelligence and Databases*, pages 60–68. Morgan Kaufmann Publishers, 1989. doi: 10.1016/B978-0-934613-53-8.50009-1.

Douglas Crockford. The application/json media type for Javascript object notation (JSON). RFC 4627, 2006.

Jianyong Dai, Ratan Guha, and Joohan Lee. Efficient virus detection using dynamic instruction sequences. *Journal of Computers*, 4(5):405–414, 2009. URL `http://www.jcomputers.us/index.php?m=content&c=index&a=show&catid=104&id=1713`.

Alan M. Davis, Edward H. Bersoff, and Edward R. Comer. A strategy for comparing alternative software development life cycle models. *IEEE Transactions on Software Engineering*, 14(10):1453–1461, 1988. doi: 10.1109/32.6190.

Saumya K. Debray, Nai-Wei Lin, and Manuel Hermnegildo. Task granularity analysis in logic programs. *ACM SIGPLAN Notices*, 25(6):174–188, 1990. doi: 10.1145/93548.93564.

Giovanni Denaro and Leonardo Mariani. Towards testing and analysis of systems that use serialization. *Electronic Notes in Theoretical Computer Science*, 116:171–184, 2005. doi: 10.1016/j.entcs.2004.02.075.

Reinhard Diestel. *Graph Theory*. Springer, 2017. ISBN 9783662536223.

Sergey N. Dorogovtsev, José Fernando F. Mendes, and Alexander N. Samukhin. Structure of growing networks with preferential linking. *Physical Review Letters*, 85(21):4633–4636, 2000. doi: 10.1103/PhysRevLett.85.4633.

Korry Douglas. *PostgreSQL: A Comprehensive Guide to Building, Programming, and Administering PostgresSQL Databases*. Sams, Indianapolis, Ind, 2003. ISBN 9780735712577.

Eleni Drinea, Mihaela Enachescu, and Michael Mitzenmacher. Variations on random graph models for the web. Technical Report TR-06-01, Computer Science Group, Harvard University, 2001.

Robert Dyer. *Bringing Ultra-Large-Scale Software Repository Mining to the Masses with Boa*. PhD thesis, Iowa State University, 2013.

Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 3rd edition, 1999. ISBN 9780805317558.

Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21, 2008. doi: 10.1016/j.entcs.2008.06.039.

Wolfgang Emmerich. Software engineering and middleware: A roadmap. In *Proceedings of he Future of Software Engineering*, pages 117–129, 2000. doi: 10.1145/336512.336542.

Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5(1):17–60, 1960.

Illés J Farkas, Imre Derényi, Albert-László Barabási, and Tamas Vicsek. Spectra of "real-world" graphs: Beyond the semicircle law. *Physical Review E*, 64(2):026704, 2001. doi: 10.1103/PhysRevE.64.026704.

Norman E. Fenton and Martin Neil. Software metrics: A roadmap. In *Proceedings of the Future of Software Engineering*, pages 357–370, 2000. doi: 10.1145/336512.336588.

Sara Fioravanti, Simone Mattolini, Fulvio Patara, and Enrico Vicario. Experimental performance evaluation of different data models for a reflection software architecture over NoSQL persistence layers. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pages 297–308, 2016. doi: 10.1145/2851553.2851561.

Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the International Conference on Management of Data*, pages 1433–1445, 2018. doi: 10.1145/3183713.3190657.

Sadayuki Furuhashi. Messagepack, 2013. URL `https://msgpack.org`.

Federico Giaimo, Hugo Andrade, Christian Berger, and Ivica Crnkovic. Improving bandwidth efficiency with self-adaptation for data marshalling on the example of a self-driving miniature car. In *Workshop Proceedings of the European Conference on Software Architecture*, 2015. doi: 10.1145/2797433.2797454.

Valeria Guimaraes, Fernanda Hondo, Rodrigo Almeida, Harley Vera, Maristela Holanda, Aleteia Araujo, Maria Emilia Walter, and Sergio Lifschitz. A study of genomic data provenance in NoSQL document-oriented database systems. In *Proceedings of the IEEE International Conference on Bioinformatics and Biomedicine*, pages 1525–1531, 2015. doi: 10.1109/BIBM.2015.735990.

Cornelia Győrödi, Robert Győrödi, George Pecherle, and Andrada Olah. A comparative study: MongoDB vs. MySQL. In *Proceedings of the International Conference on Engineering of Modern Electric Systems*, pages 1–6. IEEE, 2015. doi: 10.1109/EMES.2015.7158433.

Aric Hagberg, Pieter Swart, and Daniel Schult. Exploring network structure, dynamics, and function using NetworkX. Technical report, Los Alamos National Laboratory, 2008.

Dayne Hammes, Hiram Medero, and Harrison Mitchell. Comparison of NoSQL and SQL databases in the cloud. *Proceedings of the Southern Association for Information Systems*, 2014. URL `https://aisel.aisnet.org/sais2014/12`.

J. Han. Supporting impact analysis and change propagation in software engineering environments. In *Proceedings of the IEEE International Workshop on Software Technology and Engineering Practice Incorporating Computer Aided Software Engineering*, 1997. doi: 10.1109/step.1997.615479.

Jing Han, Ee Haihong, Guan Le, and Jian Du. Survey on NoSQL database. In *Proceedings of the International Conference on Pervasive Computing and Applications*, pages 363–366. IEEE, 2011. doi: 10.1109/ICPCA.2011.6106531.

Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. How do scientists develop and use scientific software? In *Proceedings of the ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8, 2009.

James D. Herbsleb and Audris Mockus. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on Software Engineering*, 29(6):481–494, 2003. doi: 10.1109/TSE.2003.1205177.

Marjan Hericko, Matjaz B. Juric, Ivan Rozman, Simon Beloglavec, and Ales Zivkovic. Object serialization analysis and comparison in Java and .NET. *ACM SIGPLAN Notices*, 38(8):44–54, 2003. doi: 10.1145/944579.944589.

Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 1990. doi: 10.1145/93542.93574.

Waqar Hussain, Tony Clear, and Stephen MacDonell. Emerging trends for global DevOps: A New Zealand perspective. In *Proceedings of the IEEE International Conference on Global Software Engineering*, pages 21–30, 2017.

Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the International Conference on Software Engineering*, pages 672–681, 2013. doi: 10.5555/2486788.2486877.

Jeremy Kepner, Vijay Gadepally, Dylan Hutchison, Hayden Jananthan, Timothy Mattson, Siddharth Samsi, and Albert Reuther. Associative array model of SQL, NoSQL, and NewSQL databases. In *Proceedings of the IEEE High Performance Extreme Computing Conference*, pages 1–9, 2016. doi: 10.1109/HPEC.2016.7761647.

Roger Khazan, Jesse Rabek, Scott Lewandowski, and Robert Cunningham. Technique for detecting executable malicious code using a combination of static and dynamic analyses, 2005. U.S. Patent Application 10/464,828.

Jon Kleinberg. *Algorithm Design*. Pearson, Harlow, Essex, 2014. ISBN 9781292037042.

Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and Xiao Feng Wang. Effective and efficient malware detection at the end host. In *Proceedings of the USENIX Security Symposium*, pages 351–366, 2009. doi: 10.5555/1855768.1855790.

Kenji Kono and Takashi Masuda. Efficient RMI: Dynamic specialization of object serialization. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pages 308–315, 2000. doi: 10.1109/ICDCS.2000.840943.

R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal. Stochastic models for the web graph. In *Proceedings of the Symposium on Foundations of Computer Science*, pages 57–65, 2000. doi: 10.1109/SFCS.2000.892065.

Robert Lafore. *Data Structures & Algorithms in Java*. Sams, Indianapolis, Indiana, United States, 2003. ISBN 978-0672324536.

Meir M. Lehman, Juan F. Ramil, Paul D. Wernick, Dewayne E. Perry, and Wladyslaw M. Turski. Metrics and laws of software evolution: The nineties' view. In *Proceedings of the International Software Metrics Symposium*, pages 20–32. IEEE, 1997. doi: 10.1109/METRIC.1997.637156.

Kazuaki Maeda. Comparative survey of object serialization techniques and the programming supports. *International Journal of Computer and Information Engineering*, 5(12):1488–1493, 2011. doi: 10.5281/zenodo.1084306.

Kazuaki Maeda. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In *Proceedings of the International Conference on Digital Information and Communication Technology and Its Applications*, pages 177–182, 2012. doi: 10.1109/DICTAP.2012.6215346.

Bruce McCorkendale, Xue Feng Tian, Sheng Gong, Xiaole Zhu, Jun Mao, Qingchun Meng, Ge Hua Huang, and Wei Guo Eric Hu. Systems and methods for combining static and dynamic code analysis, 2014. U.S. Patent 8,726,392.

John A. McDermid and Keith H. Bennett. Software engineering research: A critical appraisal. *IEE Proceedings—Software*, 146(4):179–186, 1999. doi: 10.1049/ip-sen:19990618.

Robert E. McGrath. Xml and scientific file formats. In *Proceedings of the Seattle Annual Meeting*, 2003.

Hao Men. *Fast and Scalable Change Propagation through Context-Insensitive Slicing*. PhD thesis, University of Calgary, 2018.

Donald Michie. "memo" functions and machine learning. *Nature*, 218(5136):19–22, 1968. doi: 10.1038/218019a0.

Justin J. Miller. Graph database applications and concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference*, 2013. URL `https://aisel.aisnet.org/sais2013/24`.

A. B. M. Moniruzzaman and Syed Akhter Hossain. NoSQL database: New era of databases for big data analytics—classification, characteristics and comparison. arXiv preprint arXiv:1307.0191, 2013.

Lea Morschel, Dmitry Litvintsev, A Rossi, Juergen Starek, Lea Morschel, Marina Sahakyan, Olufemi Adeyemi, and Vincent Garonne. dCache: Efficient message encoding for inter-service communication. Technical report, Fermi National Accelerator Laboratory, 2019.

Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the International Conference on Software Engineering*, pages 580–586, 2005. doi: 10.1109/ICSE.2005.1553604.

Iulian Neamtiu, Jeffrey S. Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 1–5, 2005. doi: 10.1145/1082983.1083143.

Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of JSON and XML data interchange formats: A case study. In *Proceedings of the ISCA International Conference on Computer Applications in Industry and Engineering*, pages 157–162, 2009.

Gregg Petri. A comparison of Oracle and MySQL. *Select Journal*, 1, 2005.

Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000. doi: 10.1002/1096-9128(200005)12:7<495::AID-CPE496>3.0.CO;2-W.

Oula Piisalo. *Serialisointi ja tallentaminen Protobuf-net-formaatilla C#-pohjaisessa pelissä*. PhD thesis, Tampere University of Applied Sciences, 2016. URL `http://urn.fi/URN:NBN:fi:amk-2016092914698`.

Robert Poljak, Patrizia Poščić, and Danijela Jakšić. Comparative analysis of the selected relational database management systems. In *Proceedings of the International Convention on Information and Communication Technology, Electronics and Microelectronics*, pages 1496–1500. IEEE, 2017. doi: 10.23919/MIPRO.2017.7973658.

Sharvari Rautmare and D. M. Bhalerao. MySQL and NoSQL database comparison for IoT application. In *Proceedings of the IEEE International Conference on Advances in Computer Applications*, pages 235–238, 2016. doi: 10.1109/ICACA.2016.7887957.

William E. Riddle and Richard E. Fairley. *Software Development Tools*. Springer, 2012. ISBN 9783540103264. doi: 10.1007/978-3-642-46430-0.

Daniel A. Schult and P. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the Python in Science Conference*, volume 2008, pages 11–16, 2008. URL `http://conference.scipy.org/proceedings/scipy2008/paper_2/`.

Michael I. Schwartzbach. Lecture notes on static analysis. Basic Research in Computer Science, University of Aarhus, Denmark, 2008.

Edwin Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, 2003. doi: 10.1109/MS.2003.1231147.

Yakov Shafranovich. *Common format and MIME type for comma-separated values (CSV) files*, 2005. RFC 4180.

Baji Shaik and Avinash Vallarapu. PostgreSQL architecture. In *Beginning PostgreSQL on the Cloud*, pages 33–61. Springer, 2018. doi: 10.1007/978-1-4842-3447-1_2.

Vatika Sharma and Meenu Dave. SQL and NoSQL databases. *International Journal of Advanced Research in Computer Science and Software Engineering*, 2(8):20–27, 2012. URL `https://www.researchgate.net/publication/303856633_SQL_and_NoSQL_Databases/link/5758557f08ae9a9c954a7573/download`.

Keshav Sud, Pakize Erdogmus, and Seifedine Kadry. *Introduction to Data Science and Machine Learning*. IntechOpen Limited, 2020. ISBN 9781838803346. doi: 10.5772/intechopen.77469.

Audie Sumaray and S. Kami Makki. A comparison of data serialization formats for optimal efficiency on a mobile platform. In *Proceedings of the International Conference on Ubiquitous Information Management and Communication*, pages 1–6, 2012. doi: 10.1145/2184751.2184810.

Clarence J. M. Tauro, N. Ganesan, Saumya Mishra, and Anupama Bhagwat. Object serialization: A study of techniques of implementing binary serialization in C++, Java and .NET. *International Journal of Computer Applications*, 6(45):25–29, 2012. doi: 10.5120/6785-9088. URL `https://www.ijcaonline.org/archives/volume45/number6/6785-9088`.

Jan Vanura and Pavel Kriz. Perfomance evaluation of Java, JavaScript and PHP serialization libraries for XML, JSON and binary formats. In *Services Computing*, pages 166–175. Springer, 2018. doi: 10.1007/978-3-319-94376-3_11.

Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. A comparison of a graph database and a relational database: A data provenance perspective. In *Proceedings of the ACM Southeast Regional Conference*, pages 1–6, 2010. doi: 10.1145/1900008.1900067.

Jim Waldo. Remote procedure calls and Java remote method invocation. *IEEE Concurrency*, 6(3):5–7, 1998. doi: 10.1109/4434.708248.

Phyllis Wan, Tiandong Wang, Richard A. Davis, and Sidney I. Resnick. Fitting the linear preferential attachment model. *Electronic Journal of Statistics*, 11(2):3738–3780, 2017. doi: 10.1214/17-EJS1327.

Andy Wigley, Mark Sutton, Stephen Wheelwright, Robert Burbidge, and Rory MacLoed. *Microsoft .NET Compact Framework (Core Reference)*. O'Reilly, 2009. ISBN 9780735637542.

Hyrum K. Wright, Miryung Kim, and Dewayne E. Perry. Validity concerns in software engineering research. In *Proceedings of the FSE/SDP Workshop on the Future of Software Engineering Research*, pages 411–414, 2010. doi: 10.1145/1882362.1882446.

Xin Zhou, Yuqin Jin, He Zhang, Shanshan Li, and Xin Huang. A map of threats to validity of systematic literature reviews in software engineering. In *Proceedings of the Asia–Pacific Software Engineering Conference*, pages 153–160. IEEE, 2016. doi: 10.1109/APSEC.2016.031.