Universitat Politècnica de Catalunya

Department d'Arquitectura de Computadors

Computer-Language based Data Prefetching Techniques

Rizkallah Touma

Advisors:

Dr. Anna Queralt Dr. Toni Cortes

A dissertation submitted in fulfillment of the requirements
for the degree of
Doctor from the Universitat Politècnica de Catalunya
in the
Doctoral Program of the Department d'Arquitectura de Computadors

Barcelona, November 26, 2018

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Acknowledgements

I would like to express my sincere gratitude to my supervisors Dr. Anna Queralt and Dr. Toni Cortes for their continuous supervision and their patience. Thank you for believing in my work even when I didn't.

I want to thank Dr. David Carrera, Dr. Gabriel Antoniu and Prof. André Brinkmann for dedicating the time to evaluate my thesis and be part of my defense committee.

I would also like to thank Prof. María S. Pérez from the Ontology Engineering Group (OEG) at the Universidad Politécnica de Madrid for her insightful feedback and guidance through the different stages of the thesis, as well as for giving me the opportunity to do my secondment in the OEG. I also want to thank Dr. Mariano Rico for his commitment to the success of my collaboration with the group.

Many thanks to my colleagues in the Storage Systems Group at the Barcelona Supercomputing Center; Jonathan Martí, Alex Barceló, Pierlauro Sciarelli, Enrico La Sala, Georgios Koloventzos, and especially Dani Gasull, for their friendship and their help in implementing the work presented in this thesis.

I am eternally grateful to my parents, Kamal and Nayla, who have taught me that I can accomplish great things in life through hard work and perseverance. You have always been a source of inspiration to me.

Last but not least, I thank Alex for being there for me and providing me with the much-needed emotional support to complete this journey. I am truly blessed to have you in my life.

This work has been supported by the European Union's Horizon 2020 research and innovation program under the BigStorage European Training Network (grant H2020-MSCA-ITN-2014-642963), the Spanish Ministry of Science and Innovation (contract TIN2015-65316) and the Generalitat de Catalunya (contract 2014-SGR-1051)

Abstract

Data prefetching has long been used as a technique to improve access times to persistent data. It is based on predicting which data records are relevant to future requests and retrieving them from persistent storage to main memory before they are needed. Data prefetching has been applied to a wide variety of persistent storage systems, from file systems to Relational Database Management Systems and NoSQL databases, with the aim of reducing access times to the data maintained by the system and thus improve the execution times of the applications using this data.

However, most existing solutions to data prefetching have applied predictions based on information that can be retrieved from the storage system itself, whether in the form of heuristics based on the data schema or data access patterns detected by monitoring access to the system. There are multiple disadvantages of these approaches in terms of the rigidity of the heuristics they use, the accuracy of the predictions they make and / or the time they need to make these predictions, a process often performed while the applications are accessing the data and causing considerable overhead.

In light of the above, this thesis proposes two novel approaches to data prefetching based on predictions made by analyzing the instructions and statements of the computer languages used to access persistent data. The proposed approaches take into consideration *how* the data is accessed by the higher-level applications, make accurate predictions and are performed without causing any additional overhead. The first of the proposed approaches aims at analyzing instructions of applications written in object-oriented languages in order to prefetch data from Persistent Object Stores. On the other hand, the second approach analyzes statements and historic logs of the declarative query language SPARQL in order to prefetch data from RDF Triplestores.

Keywords: Data Prefetching, Persistent Object Stores, Object-Oriented Languages, RDF Triplestore, SPARQL

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In spite of the recent technological advances in computer science, access to disk is still the bottleneck in many computer applications that require persistent data. Between 1980 and 2000, microprocessor performance has improved at an average rate of 60% per year. By contrast, disk access speeds have only experienced a 10% improvement per year over the same period [69, 83]. This growing gap between processing speed and the speed of access to the data needed by the applications has produced significant amount of research in methods to minimize data access times.

In the field of databases and data storage systems, two methods to improve this access time have prevailed. The first of these methods is caching, which is based on the idea of keeping recently retrieved data in memory cache for faster access with subsequent requests. However, this solution only works if the exact same data is accessed multiple times. In reality, it is more common to have consecutive requests that access several different, albeit related, data records [26, 72].

The second method that was proposed in order to deal with this issue is data prefetching. **Data Prefetching is defined as retrieving data records from persistent storage to main memory in anticipation of later use.** Unlike caching, prefetching aims to predict additional data records that are likely to be accessed by subsequent requests and retrieve them before they are needed. In order to predict which data records should be prefetched, several approaches have been studied.

The first such approach is based on **the schema of the data**. This approach analyzes the schema of the data being manipulated and predicts which data records should be prefetched based on the relations found in the schema. Given that the data schema is predefined in most database applications, the schema analysis is simple, is only performed once and does not need to be modified. However, this also means that this type of approaches does not take into consideration the different applications accessing the data nor the actual values of the data records.

Another approach to data prefetching was put forward based on monitoring **access to the data records** and detecting recurring access patterns. The detected patterns are then used to predict which data records should be prefetched. However, this monitoring process needs to take place while applications are being executed and accessing data. Thus, it can add a non-negligible overhead to application execution time and/or consume a considerable amount of memory.

The third and least-studied approach tries to tackle these limitations by basing the predictions on the **computer language instructions and statements** used to access the data. The reasoning behind this approach is that it can lead to more accurate prediction, given that it takes into account *how* the data is accessed by the higher-level program code. Moreover, if needed, the prediction can be improved by studying historical executions and trying to detect repeated execution patterns, which in turn indicate which data records are going to be accessed.

In view of the above, this thesis investigates approaches to data prefetching based on predictions obtained from analyzing the instructions and statements of computer languages. In particular, we develop two such approaches using two different types of languages: object-oriented programming languages and declarative query languages.

First, we develop an approach that analyzes the code of applications written in object-oriented languages, defined in Section 1.1.1, and predicts which data objects are accessed by the application. The approach is based on static code analysis that is done prior to the application execution and hence does not add any overhead. We also propose various strategies to deal with cases that require runtime information unavailable prior to the execution of the application. Moreover, we integrate this code analysis approach into a Persistent Object Store, defined in Section 1.1.3, to perform data prefetching based on the predictions made by the approach.

Afterwards, we propose a second approach that analyzes the statements of declarative query languages, defined in Section 1.1.2. We use as a case study the query language SPARQL and develop an approach to predict data to be prefetched by analyzing historic SPARQL query logs and detecting recurring query patterns. Furthermore, we design a prefetching and caching system to be integrated into RDF Triplestores, defined in Section 1.1.4, in order to prefetch the predicted data.

1.1 Background

This thesis draws concepts from various areas in the fields of Computer Languages and Persistent Storage. As such, we present in this section an overview of object-oriented Languages (Section 1.1.1), Declarative Query Languages, Persistent Object Stores (Section 1.1.3) and RDF Triplestores (Section 1.1.4).

1.1.1 Object-Oriented Programming Languages

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects", which may contain data, in the form of *attributes*; and code, in the form of *methods* [49]. The object's methods can access and modify the object's data attributes. OOP languages are class-based, meaning that objects in those languages are instances of classes, which also determine their type.

For example, Figure 1.1 shows the program code of an application written in Java, an OOP language. In this example, we can see the different classes, i.e. object types, defined by the application, such as the class *Transaction* or the class *Employee*. We can also distinguish two types of attributes in these classes: attributes of primitive types and attributes of user-defined types.

Attributes of primitive types, such as *Integer*, *String* or *Date*, contain the data of each object of the class. For instance, the class *Employee* in Figure 1.1 has the field *salary* of the primitive type *Integer* and the field *dateOfBirth* of the primitive type *Date*. On the other hand, a class may also have attributes of complex types that represent relationships between two different types. A relationship from type *t* to type *t'* is represented by an attribute of type *t'* defined in type *t*. For instance, the relationship

```java
public class Transaction {
 private Account account;
 private Employee emp;
 private TransactionType type;
 ...
}

public class Employee {
 private Company company;
 private String name;
 private Integer salary,
 private Date dateOfBirth;
 ...
}

public class BankManagement {
 private ArrayList<Transaction> transactions;
 private Customer manager;

 public void setAllTransCustomers() {
   for (Transaction trans : this.transactions) {
     trans.getAccount().setCustomer(this.manager);
   }
 }
}

...
```

Figure 1.1: *Example of program code written in an object-oriented language (Java).*

from the type *Transaction* to the type *Account* is implemented in Figure 1.1 by an attribute of type *Account* defined inside the type *Transaction*.

Finally, Figure 1.1 also contains the method *setAllTransCustomers()* defined in the class *BankManagement*. This method can access, and manipulate, the attributes of the object to which it belongs. Furthermore, it can invoke other methods on the objects related to its owning object.

1.1.2 Declarative Query Languages

Query languages are computer languages used to make queries in database and information systems. Most modern query languages follow the Declarative Programming (DP) paradigm, which means that they express the logic of a computation without describing the exact instructions to be executed. That is, DP languages describe *what*

```
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT * WHERE {
   dbr:Iker_Casillas  dbo:formerTeam  ?team .
}
```

Figure 1.2: *Example code written in a declarative query language (SPARQL).*

the program must accomplish in terms of the problem domain, rather than *how* to accomplish it as a sequence of explicit steps.

Some examples of database query languages that follow the DP paradigm include SQL, XQuery and SPARQL. Figure 1.2 shows an example query written in SPARQL query language. SPARQL is a high-level query language used to access data stored in the RDF format (explained in detail in Section 1.1.4). In the example in Figure 1.2, the query asks for entities related to the entity *dbr:Iker_ Casillas* through the property *dbo:formerTeam*.

Regardless of the details of the language, this example shows that declarative query languages do not use specific statements or steps that should be followed, but rather describe what should be done. In the case of the query in Figure 1.2, the query asks for the former teams of *Iker Casillas*.

1.1.3 Persistent Object Stores

Persistent Object Stores (POSs) are data storage systems that record and retrieve persistent data in the form of complete objects [13]. In this context, an **object** is defined as an instance of a particular type defined by the schema of the POS. The object consists of a combination of fields, which can either be of primitive types (e.g. *String*, *Integer*, *Float*), or of another type defined by the schema, in which case they represent a relationship between two objects.

POSs were developed to avoid the impedance mismatch that occurs when developing object-oriented applications on top of Relational Database Management Systems (RDBMSs). This impedance mismatch is caused by the conceptual and technical difficulties in mapping application data, in the form of objects, to persistent data stored

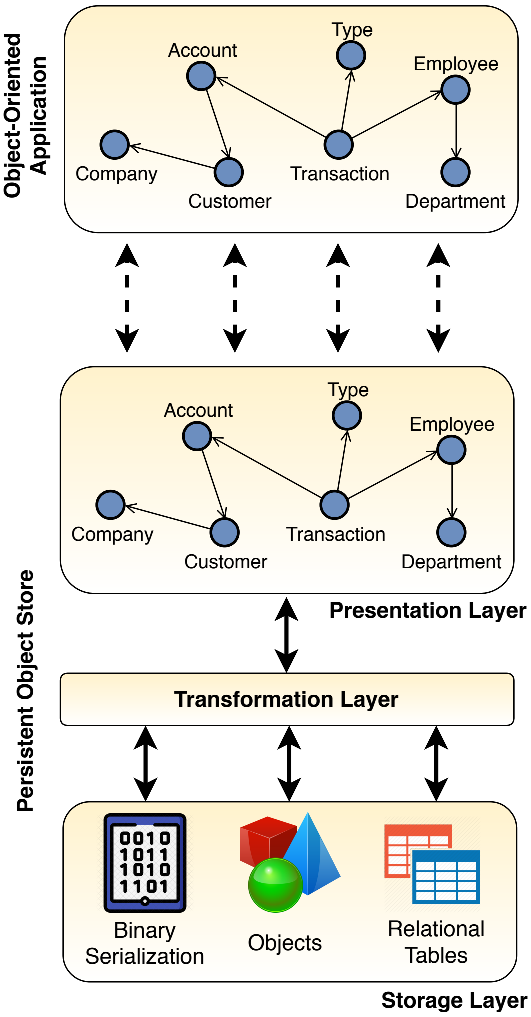


Figure 1.3: *Structure of a generic Persistent Object Store (POS). The used storage mechanism depicted in the storage layer depends on the type of the POS.*

in the RDBMS in the form of tables. By contrast, using a POS the application objects can be directly mapped to objects stored in the POS.

Figure 1.3 shows the architecture of a generic POS and how it can be used by an OO application to solve this impedance mismatch. The top layer in the figure shows the objects manipulated by the OO application. In order to store or retrieve the objects, the application communicates with the *presentation layer* of the POS, without the need to perform any type of transformation on the objects. Inside the POS, this layer then communicates with the *transformation layer*, which proceeds to transform the objects into a format suitable for persistent storage (e.g. serializing the objects, converting them to relational tables, etc). Finally, the *transformation layer* sends the transformed objects to the *storage layer*, which stores the objects in one of various storage mechanisms, depending on the type of the POS.

The underlying storage mechanism of a POS varies from simple binary serialization of objects to complex Object-Oriented Databases (OODB), which store objects directly onto disk, and Object-Relational Mapping Systems (ORM), which internally transform objects into tables and store them in an RDBMS. The most popular OODBs include Caché and Versant[1], while some of the most used ORMs include Hibernate, Apache OpenJPA and Data Nucleus[2]. The rise of NoSQL databases has also led to the development of object mapping systems for other types of databases such as Neo4J's Object-Graph Mapping (OGM)[3].

Regardless of their internal storage mechanism, POSs provide a conceptual abstraction for mapping database records to objects in object-oriented languages. This abstraction avoids the impedance mismatch that occurs with other types of databases and makes it easier to access persistent data without having to worry about database access and query details, which amount to 30% of the total code of an application according to previous studies [4] [20]. **Moreover, this symmetry between OOP programming languages and the underlying storage opens the door to new approaches when it comes to detecting data access patterns.**

1.1.4 RDF Triplestores

The **Resource Description Framework (RDF)** is a simple, extensible graph data model for representing information on the web [81]. Its main structure consists of *triples* that link two resources; called the *subject* and *object* of the triple, through a property; called the *predicate* of the triple.

Figure 1.4 shows an example RDF data graph. In this graph, we can identify four different resources: *Iker Casillas*, *Real Madrid*, *Zinedine Zidane* and *Spain* and three different properties: *was born in*, *played for* and *is managed by*. The triples present in

[1] Caché: `http://www.intersystems.com/our-products/cache/cache-overview/`, Versant: `http://www.actian.com/products/operational-databases/versant/`

[2] Hibernate: `http://hibernate.org/`, Apache Open JPA: `http://openjpa.apache.org/`, Data Nucleus: `http://www.datanucleus.org/`

[3] Neo4J OGM: `https://neo4j.com/docs/ogm-manual/current/`

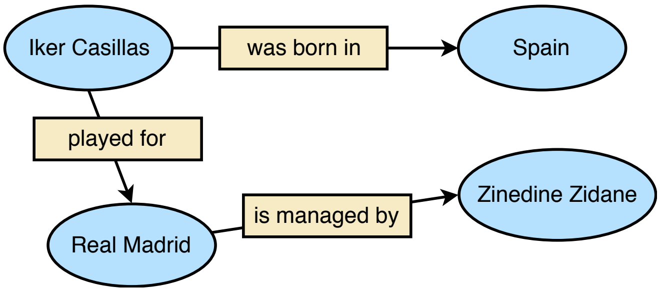


Figure 1.4: *RDF data graph containing resources (blue circles) and properties (yellow rectangles).*

this RDF graph are the ones resulting from a property linking two different resources. For example in the triple *Iker Casillas played for Real Madrid*, we have the predicate *played for* linking the object of the triple *Iker Casillas* to the subject *Real Madrid*.

The importance of RDF comes from this triple-based structure, which gives it great flexibility in representing semantic information in a machine-understandable way. Moreover, RDF graphs do not follow a fixed, predefined schema and hence are easily extensible with new information. These advantages made RDF the *de facto* standard for publishing data within the Linked Data project [7].

An **RDF Triplestore** is a purpose-build database for the storage and retrieval of RDF data graphs through semantic queries. RDF triplestores are a cornerstone of Linked Data, the standard for publishing structured data on the Semantic Web [7], which has grown to provide a wealth of publicly-available data, with some repositories containing millions of concepts described by RDF triples (e.g. DBpedia[4], FOAF[5], GeoNames[6]).

Figure 1.5 shows the structure of an RDF triplestore. An application can access the data in these triplestores by issuing semantic queries using SPARQL, the standard, high-level, declarative query language for RDF stores. The SPARQL queries are received by the triplestore's **SPARQL Endpoint**, which proceeds to execute the queries on the triplestore's data and returns the corresponding results in RDF format.

For instance, in the previous section Figure 1.2 shows an example SPARQL query that asks for the former teams of the player *Iker Casillas*. If we evaluate this query against the data graph shown in Figure 1.4, it would return the resource *Real Madrid*.

[4] DBpedia: https://wiki.dbpedia.org/

[5] FOAF: http://www.foaf-project.org/

[6] GeoNames: http://www.geonames.org/

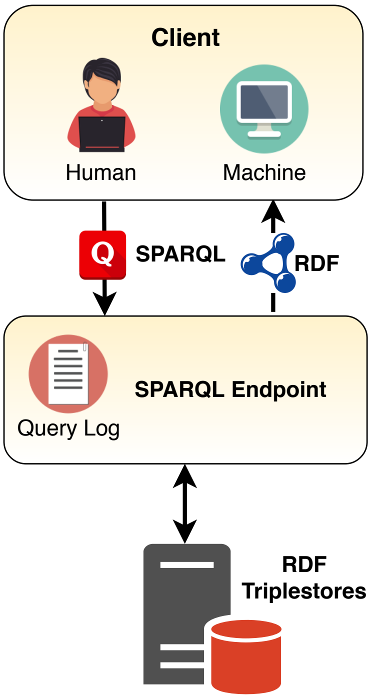


Figure 1.5: *Structure of an RDF Triplestore and its SPARQL endpoint.*

SPARQL endpoints of popular Linked Data repositories, such as DBpedia, often need to execute a big number of queries coming at high-frequency from multiple clients. The increasing workload that these SPARQL endpoints face can result in high query response times which negatively influence the user experience when accessing the data of the triplestore [54, 84]. However, **SPARQL endpoints also tend to keep a log of all the queries received from clients, which can be used to detect repetitive access patterns to the triplestore's data.**

1.2 Problem Statement

In general, regardless of the used computer language and persistent storage system, access times to data dominate the execution time of applications. This is due to the fact that access to persistent storage is orders of magnitude slower than CPU execution time. The exact latency of this access depends on the type of medium used to store the persistent data. For instance, accessing data on HDD is 10^7 slower than executing a CPU instruction while for SSD this number stands at 10^5 and for NVM (Non-Volatile Memory) it is estimated to be around 10^2 times slower [83].

This latency can be mitigated by keeping the data that is likely to be accessed by an application in the near future in main memory (i.e. RAM). Reading data from RAM is 10^3 to 10^5 times faster than reading data from persistent storage, due in part to the fact that RAMs allow data items to be read or written in almost the same amount of time irrespective of the physical location of data inside the memory. However, main memory is usually smaller in size than persistent storage and can only hold a fraction of the data kept on persistent storage. Moreover, main memory is volatile and cannot be used to maintain persistent copies of data after the execution of applications is finished.

While keeping a copy of the data previously accessed by an application in main memory might help when the same data is repeatedly accessed, a more effective solution would be to predict which data an application is going to access and prefetch this data from persistent storage before it is needed by the application. Prefetching has long been studied as an important approach to improve access times to data. **However, current approaches to prefetching do not always take full advantage of the information that can be retrieved from the different layers of the technological stack to make predictions on which data should be prefetched.**

For instance, previous approaches to prefetching in Persistent Object Stores either use fixed heuristics based on the schema of the data or try to detect data access patterns by monitoring application execution. These approaches are based on a most-common case scenario, which does not always provide accurate prediction, and are done during application execution, which might cause overhead and / or consume additional memory. Other approaches base the prefetching on predictions made by analyzing the code of the OO applications that access the POS, but these have been largely theoretical without any in-depth analysis of the prediction accuracy or the improvement that can be achieved in application execution time [8, 43]. Section 2.3 details previous approaches to prefetching in POSs and highlights their shortcomings.

On the other hand, given that RDF is a flexible, schema-less data format, there has been a limited amount of research into prefetching based on schema analysis of RDF Triplestores. By contrast, previous prefetching techniques in this area have been based on data retrieved from the triplestore, which is not always available across triplestores, while others aimed at using prefetching to reduce data access latency in specific graph traversal problems only. While there has been some work on prefetching data based on analysis of SPARQL queries, these approaches have been limited to prefetching the results of the most frequent previous queries or applying query augmentation based on data found in the RDF Triplestore [54, 55, 84]. A detailed overview of the related work in prefetching in RDF Triplestores is provided in Section 2.4.

1.3 Hypothesis and Proposed Solution

Based on the technologies described in Section 1.1 and the problems stated in Section 1.2, this thesis presents the following hypothesis:

In a technological setting where applications need access to persistent data, it is possible to perform data prefetching based on predictions made by analyzing the instructions and statements of the used computer language, without involving the underlying data storage system.

Figure 1.6 shows an abstraction of how this hypothesis can be tested on an application using a persistent data system. The figure indicates that, regardless of the used computer language and data store, the application communicates with the store using a set of predefined calls to store and retrieve data. Our proposed solution, shown in the blue rectangles in Figure 1.6, takes as input the computer language statements of the application, which it utilizes to perform a prediction process that generates predictions of which data should be prefetched. Afterwards, the predictions are used by a prefetching process to retrieve the data from the persistent data store into a memory cache, where it can be found by the application when it is needed. The implementation of how this process is performed depends on the used technologies.

In the case of Persistent Object Stores, the objects are retrieved by the object-oriented application directly from the store through a set of instructions that automatically convert the persistent objects to the corresponding object type defined by the application, as explained in Section 1.1.3. Thus, our proposed solution in this setting consists of applying static code analysis on the object-oriented application to predict which persistent objects are accessed. We then prefetch the predicted objects into the

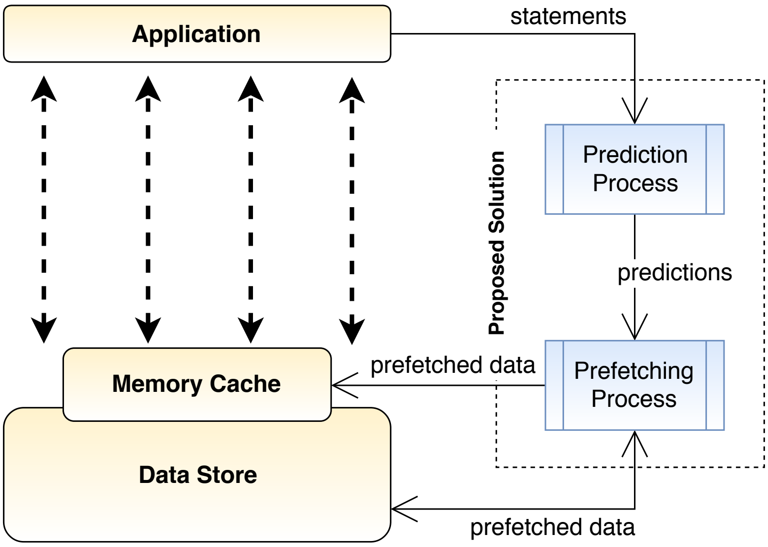


Figure 1.6: *Abstraction of our proposed solution.*

memory cache of the system, and thus the application retrieves them from the cache when they are needed.

As for RDF Triplestores, data is retrieved from the store through SPARQL queries which return data in RDF format. Unlike the case with object-oriented languages where the entire application code is known beforehand, the SPARQL queries are received consecutively and are not known in advance. Thus, instead of performing the prediction statically we needed to adopt another approach that can predict upcoming queries. We achieve this by analyzing historic query logs of SPARQL queries to detect recurring query patterns and predict which data should be prefetched.

Another difference between the implementation of the two solutions is the fact that RDF Triplestores do not have an integrated memory cache. Hence, an external memory cache should be constructed into which the predicted data is prefetched. This difference is depicted in Figure 1.6 by situating the cache in between the borders of the store, indicating that the cache could be either internal or external depending on which technological setting is used. While the approach developed in this thesis focuses on the prediction of which data should be prefetched from the RDF Triplestore, we also detail the design of a caching and prefetching system in Chapter 5 to be integrated into a Triplestore as part of the future work of the thesis.

1.4 Research Questions

We evaluated the hypothesis and solution proposed in Section 1.3 by answering the following set of research questions:

(1) In the case of Persistent Object Stores and object-oriented applications:

RQ1: What is the percentage of applications for which static code analysis can predict access to persistent objects?

RQ2: Can the proposed static code analysis be performed within a reasonable amount of time?

RQ3: What is the prediction accuracy of the proposed static code analysis?

RQ4: How much in advance can the proposed static code analysis predict access to persistent objects?

RQ5: Does the proposed prefetching approach improve application execution times?

RQ6: What is the object hit rate of the prefetching approach?

(2) In the case of RDF Triplestores and the declarative query language SPARQL:

RQ7: What is the prediction accuracy of the proposed query-log analysis approach?

RQ8: Can the predictions be made within a reasonable amount of time?

RQ9: What is the cache hit rate of the proposed prefetching approach?

1.5 Contributions

This thesis presents three novel contributions to the scientific community that aim to prove the hypothesis formulated in Section 1.3:

C1 - An approach to predict access to persistent objects by statically analyzing the source code of object-oriented applications.

In this contribution, we develop an approach that analyzes the source code of object-oriented applications that access data stored in a POS in order to predict access to persistent objects. Our approach takes advantage of the symmetry between application objects and persistent objects to perform the prediction process before the application is executed and hence does not cause any overhead. The predictions made by our approach can then be used to apply a variety of techniques that aim to improve access to persistent data, such as prefetching, cache replacement policies and dynamic data placement.

This contribution aims to answer the research questions **RQ1**, **RQ2**, **RQ3** and **RQ4**.

C2 - An approach to data prefetching for Persistent Object Stores.

We developed this contribution by integrating our static code analysis approach into an existing POS in order to prefetch the predicted persistent objects. We also optimize the prefetching approach with automatic parallelization in order to take advantage of data distribution and maximize the benefits obtained from prefetching. Moreover, we demonstrate in this contribution that prefetching data based on our approach reduces the times spent by applications waiting for data to be accessed, thus improving the overall application execution time.

This contribution aims to answer the research questions **RQ5** and **RQ6**.

C3 - A query-log analysis approach to prefetch data in RDF Triplestores.

In this contribution, we present an approach to prefetching in RDF Triplestores based on analysis of historic query logs of the SPARQL declarative query language. The approach detects recurring query patterns in previous queries and uses the detected patterns to prefetch data relevant to subsequent queries. The novelty of our approach is that we measure two independent types of similarity between queries: structural similarity and triple-pattern similarity. Using these two similarities, we apply machine learning algorithms to detect recurring patterns in the query logs and prefetch data relevant to subsequent queries.

This contribution aims to answer the research questions **RQ7**, **RQ8** and **RQ9**.

1.6 Thesis Structure

The rest of this thesis is structured as follows. Chapter 2 discusses previous approaches to prefetching in the two domains to which this thesis belongs: **(1)** Persistent Object Stores (POSs), and **(2)** RDF Triplestores, underlining their shortcomings and highlighting how our proposed approach tackles these problems.

Afterwards, Chapter 3 presents the first contribution of this thesis **(C1)** by detailing the theoretical background and formalization of the first of our proposed approaches to prefetching; static code analysis of object-oriented applications. The chapter also validates our proposed approach by analyzing the source code of a representative sample of Java applications.

Chapter 4 then presents the second contribution **(C2) by** discussing how the proposed static code analysis approach was implemented in *dataClay*, a Persistent Object Store. The chapter also offers an evaluation of performance using two benchmarks, one standard benchmark for POSs and one benchmark commonly used for Big Data applications. The experimental results demonstrate that our proposed approach can indeed improve data access times, and thus application execution times, when compared to other approaches to prefetching.

We then present the last contribution of the thesis **(C3)** in Chapter 5, which exposes our approach to prefetch data from RDF Triplestores by analyzing SPARQL query logs. The chapter also validates the proposed approach on real-world query logs and shows that it can achieve a higher cache-hit rate than previous approaches.

Finally, Chapter 6 outlines the conclusions achieved with this thesis and highlights future directions to be taken.

Chapter 2

State of the Art

Data Prefetching is defined as retrieving objects from persistent storage to main memory in anticipation of later use. Prefetching techniques are usually split into two broad categories: hardware-based and software-based [14]. In this section, we first expose a summary of the most important prefetching techniques in each of these categories before giving a detailed overview of approaches in the two domains to which this thesis belongs: **(1)** Persistent Object Stores in Section 2.3, and **(2)** RDF Triplestores in Section 2.4. We also discuss the limitations of previous approaches in both domains and highlight the benefits of our proposed approach.

2.1 Hardware-Based Prefetching

The first approaches to prefetching were based solely on computer hardware, with the aim of prefetching data from the hard disk into the embedded memory buffer of the disk. These approaches do not take into consideration any higher-level information, such as that produced by analyzing the data stores or program code.

The most basic such technique is the One Block Look-Ahead (OBL) and its variants. This technique is based on the memory addresses of the data and prefetches the block adjacent to the one currently accessed. A popular variant of this technique is the N-Blocks Look-Ahead which allows to prefetch the N blocks adjacent to the current block [67]. Another variant is presented by Baer and Chen who propose a "hardware support unit" that looks ahead in the instructions to be executed and prefetches the data from the memory addresses associated with the load and store instructions [5].

Combined caching and prefetching policies are more sophisticated hardware-based techniques that take both caching and prefetching into consideration. Several approaches have been proposed in this category, such as dividing the cache into various parts, some for accessed blocks and others for prefetched blocks and managing each part separately [45, 70]. Other proposals suggest multi-layered prefetching to manage global memory in multi-machine distributed systems [80]. For more information on combined caching and prefetching approaches, Cao *et al.* provide an extensive study on this type of techniques [16].

Most other approaches simply offer improvements on these traditional techniques, such as arranging the blocks to prefetch in order of storage on disk to reduce disk-seek times [79], or associating each access request with the ID of a "logical context", such as thread IDs, to provide context-aware prefetching [76].

2.2 Software-Based Prefetching

Software-based prefetching techniques are newer than their hardware-based counterparts and allow the programmer or compiler to insert prefetch instructions into programs. The motivation behind these strategies is the higher possibility of a compiler or developer having better knowledge of the application's data requirements, which makes it more promising in terms of prefetching accuracy [14]. Given that these approaches are performed at a higher level, they do not prefetch data into the disk buffer but rather into the database's memory cache.

The first type of software-based prefetching techniques is **history-based**; these techniques analyze the execution traces of the program in order to find data access patterns that can be used to prefetch data in subsequent executions. For instance, some approaches detect access patterns by monitoring miss addresses of the program instructions and prefetch data in future executions from these addresses [24, 65].

Another type of software-based prefetching uses **query rewriting** techniques. This type is particularly popular when dealing with relational databases, with several proposals on how relational queries can be statically moved forward in the program code or combined to prefetch the query results before they are accessed [10, 11, 73].

Several prefetching approaches **targeting specific data structures** have also been developed. These approaches try to tackle the limitations of generic prefetching approaches by offering specific improvements for the data structures that they address, such as DNA sequences [61], hash structures [19], linked list structures [18, 15, 47], pointer-based structures [56, 78] or dense matrices [63].

Similarly, while most of the prefetching techniques presented so far can be applied independently of the database type, researchers have developed other approaches that take advantage of the properties of **a specific type of databases**. Examples include prefetching techniques for in-memory key-value stores [85], graph databases [66] and document stores [41].

All of the approaches discussed in the rest of this chapter fall into this last category, they are software-based approaches designed to prefetch data from a specific type of database; either Persistent Object Stores or RDF Triplestores. For a more refined analysis, we further divide the approaches in both fields into the following categories:

- Techniques based on schema analysis,
- Techniques based on analysis of the data in the store, and
- Techniques based on analysis of the computer language used to access the data.

2.3 Prefetching in Persistent Object Stores

Perhaps the most-studied type of NoSQL databases when it comes to prefetching is Persistent Object Stores (POSs). This is due to the fact that POSs precede other NoSQL databases and that the structure in which they expose data, in the form of objects and relations between these objects, is rich in semantics that can produce detailed information and patterns about how the data is accessed.

2.3.1 Schema-Based Prefetching

The only technique to predict access to persistent objects that falls into this category is the **Referenced-Objects Predictor (ROP)**. This approach is based on the following heuristic: each time an object is accessed, all the objects referenced from it are likely to be accessed as well [21]. The technique can be applied by using different *fetch depths*, which indicate the number of object relations that should be crossed when performing the prefetching. For instance, using a *fetch depth* of 1 would only prefetch the objects directly referenced from an object o while a *fetch depth* of 2 would also prefetch the objects referenced from these objects as well, and so on.

While this approach does not always provide accurate prediction of which data should be prefetched, it is widely used in commercial POSs because it does not involve a complex and costly prediction process. Hibernate [21], Data Nucleus [23], Neo4JOGM [64] and Spring Data JPA [33] all support this technique through specific configuration settings with varying degrees of flexibility (e.g. apply the prefetching on system level or only to specific object types).

On the other hand, Han *et al.* tackle a major drawback of previous prefetching approaches in [36]; they have been usually done on object or page level meaning that they only work when the exact same object or page is repeatedly accessed. However, real-world applications tend to be repetitive on the level of the types of objects accessed. In other words, applications have patterns of the form "each time an object of type A is accessed, the referenced objects of types B and C are accessed along with it".

The authors of [36] exploit this repetitiveness by analyzing access patterns at the type-level instead of the object or page level. They apply machine-learning techniques in order to detect type-level access patterns at runtime and use the discovered patterns to prefetch objects that are predicted to be accessed next.

In an optimization of their work, Han *et al.* propose constructing a materialized database view for each detected access pattern [34]. According to the authors, this has the benefit of reducing the number of disk accesses and improves the overall performance when compared with the original type-level access patterns approach.

2.3.2 Data-Based Prefetching

The vast majority of research done in prefetching in POSs has been based on analyzing access patterns to the persistent objects. Knafla presents the first such work by using the additional information obtained through analyzing objects and their relationships with each other [50]. His approach is based on the idea that when an object a references an object b which resides in a different page than a, then this page becomes a candidate for prefetching. The approach also includes an optimization to improve the accuracy of the prefetching prediction by delaying the prefetch when an object has more than two outgoing references to objects residing in other pages.

The same author presents an extension of his work by modeling object relationships as a Discrete-Time Markov Chain and calculating the probability that a certain page will be accessed [51]. The decision to prefetch a page is based on several cost metrics that compare the benefits of a correct prefetch with the costs of an incorrect one.

Curewtiz *et al.* propose a novel approach to prefetching in POSs by using common compression algorithms [22]. In particular, the authors use three different compression algorithms, the Lempel-Zev, Prediction-by-Partial-Match and Markov-Predictor compressors, to model relations between object and predict which objects should be prefetched. They conclude that the Prediction-by-Partial-Match algorithm offers the best prefetching accuracy.

He and Marquez present a prefetching technique based on the concepts of cache and path consciousness in [39]. Their approach is based on two main ideas: dividing pages into "memory resident" and "memory non-resident" pages (cache consciousness), and storing features in the object trace during training, these features are then used to identify the current path of navigation (path consciousness).

The notion of an object's "context" was first suggested by Bernstein in [6] where an object is loaded as a predictor of future accesses. A context in this case can be a stored collection of relationships, a query result or a complex object with compositions. This context is then used when a new access to the main object is made and, for instance, if some attribute a of an object is accessed, the system prefetches attribute a for all objects in the accessed object's context.

Garbatov *et al.* use stochastic methods to analyze the runtime behavior of object-oriented applications in order to predict object accesses of future executions in [31]. The approach automatically modifies the Java bytecode of the applications in order to mark the start and end of each context (in this case, a class method).

2.3.3 Computer Language-Based Prefetching

Using static code analysis to prefetch persistent objects was first suggested by Blair *et al.* The approach analyzes the program code of OO applications at compile-time in order to model object relations and detect when the invocation of a method causes access to a different page [8]. This information is then used at runtime in order to prefetch the page once the execution of the method starts.

Ibrahim and Cook [43] propose AutoFetch, a tool that automatically profiles the traversals resulting from queries made to a POS, and uses this information to calculate a traversal profile. This profile is then used to predict future traversals and augment queries with the prefetch specification. The profile is generated at runtime and the queries are classified using their call-stack and the query string.

2.3.4 Other Types of Approaches

Ahn *et al.* present an interesting approach based on prefetching objects from a set of selected candidate pages [1]. The main distinctive feature of this approach is that, while being oriented towards POSs, it only prefetches objects from selected candidate pages without using any object semantics. The authors argue that this technique is easier to implement and less intrusive than keeping track of object semantics and relationships, which is followed by most other approaches. The algorithm is explained in detail in [2].

Finally, some commercial POSs, such as Django [27], allow the developer to manually supply prefetching hints by using predefined prefetching instructions that the developer needs to explicitly invoke with each access to the POS. This manual specification might result in more accurate prefetching, given that the developer has better knowledge of the data accesses of the application, but it is a tedious task that requires manual inspection of the entire application code. Moreover, correct prefetching hints are difficult to determine and incorrect ones are hard to detect [43].

For more information, [52] includes an extensive, albeit outdated, survey of different prefetching techniques while both [32] and [8] present taxonomies categorizing prefetching techniques in object-oriented databases.

Table 2.1: *Taxonomy of prefetching techniques in Persistent Object Stores*

Ref.	Prediction Technique	Prediction Time	Prediction Level	Prefetching Granularity
[21]	Schema-based	Compile-time	Type	Object
[36]	Schema-Based	Runtime	Type	Object
[34]	Schema-Based	Runtime	Type	Object
[50]	Data-Based	Runtime	Object	Page
[51]	Data-Based	Runtime	Object	Page
[22]	Data-Based	Runtime	Object	Page
[6]	Data-Based	Runtime	Object	Object, Attribute
[39]	Data-Based	Runtime	Object	Page
[31]	Data-Based	Runtime	Object	Page
[8]	Computer Language-Based	Compile-time	Type	Page
[43]	Computer Language-Based	Runtime	Object	Page
[2]	*Other*	Runtime	Type	Page

2.3.5 Taxonomy

This section presents a taxonomy of prefetching approaches targeted towards POSs. We based the categorization on the taxonomies included in [32] and [8], modifying them to accommodate new techniques and approaches that were not considered. Moreover, we added a new dimension that includes the categories discussed in Section 2.2.

Table 2.1 shows the taxonomy including the approaches previously discussed in this section. The dimensions according to which the approaches were categorized are:

- **Prediction technique:** indicates whether the prediction is schema-based, data-based or computer language-based.
- **Prediction time:** indicates whether the prediction is done before application execution (i.e. compile-time) or during application execution (i.e. runtime).
- **Prediction level:** indicates whether the prediction is made at the object-level or type-level. While all schema-based techniques do the prediction at the type-level, computer language-based techniques may use either object-level prediction (e.g. [43]) or type-level prediction (e.g. [8]).
- **Prefetching granularity:** indicates whether the approach prefetches objects or entire pages of objects from persistent storage. In a minority of cases, specific attributes of an object can be prefetched separately.

2.3.6 Discussion

Following the taxonomy presented in Table 2.1, our approach would fall into the categories of computer language-based, compile-time, type-level prediction, with an object-level prefetching granularity. The benefits of using **computer language-based** prediction for prefetching are highlighted by Gerlhof *et al.*, who provide a quantitative comparison between a runtime predictor and a computer language-based technique and conclude that static computer language-based techniques are a promising alternative to expensive monitoring-based predictors [32].

Moreover, using computer language-based prediction allows us to take into account information about how applications access the data without needing runtime monitoring of application execution, which is not possible to do at the store-level. Finally, in the cases where various applications access the same store, computer language-based techniques also allow to provide more accurate, application-specific prediction on which objects should be prefetched.

The advantage of performing the prediction process at **compile-time** is the absence of overhead present in techniques which need information gathered at runtime, which can amount to roughly 10% of the execution time [31]. Similarly, performing the prediction at the **type-level** has more advantages than doing the process at the object-level since it does not store the information for each individual object, thus reducing the amount of memory needed [36]. Moreover, it can capture access patterns occurring even when different objects of the same type are accessed [35]. Finally, prefetching **individual objects** instead of entire pages of objects reduces the amount of memory occupied by other objects in the same page that will not necessarily be accessed.

Table 2.1 shows that the most similar work to the approach proposed in this thesis is the approach presented in [8], which uses static code analysis to perform the prediction process at compile-time. However, this work does not offer any in-depth analysis of common code constructs, such as loop and branching statements. Moreover, the authors do not provide any implementation or evaluation of the proposed approach. Finally, the proposal in [8] prefetches entire pages of objects at once, which is far less accurate than our approach of prefetching individual objects.

2.4 Prefetching in RDF Triplestores

RDF Triplestores are a much newer technology than POSs and hence there has been less research in prefetching in this domain. Given the structure of the RDF data graph and the fact it is a schema-flexible data format, most previous approaches have been data-based, analyzing the access patterns to the resources in the triplestore and modeling the relationships between them.

2.4.1 Schema-Based Prefetching

Similarly to Persistent Object Stores, prefetching approaches based on schema analysis have not received considerable amount of research. An example of a schema-based approach to prefetching in RDF triplestores is the one presented by Ding *et al.*, which uses application-specific schema design to store large RDF graphs [25]. The authors also claim that the discovery of subject-property query sequences aids in the design of prefetching strategies. However, they do not go into any further details about how the prefetching approach can be designed or implemented.

2.4.2 Data-Based Prefetching

The first approaches to data-based prefetching in RDF Triplestores offered simple solutions that aim at improving the execution times of specific applications and use cases. One such approach is presented by Gao *et al.*, who propose prefetching a predetermined set of path expressions while solving multi-source multi-destination (MSMD) problems on RDF graphs [30]. They evaluate their prefetching technique and conclude that a more optimal prefetching strategy is needed in the future, given that their technique does not always improve performance.

Hartig *et al.* propose a more general approach that dereferences resource URIs as soon as the URI becomes part of a query solution [38]. This is in contrast with the general practice of only deferencing URIs at the time when the corresponding RDF graph is needed. Another approach is proposed by Pan *et al.*, who assign weights to different triple patterns based on the occurrence frequency of each triple pattern in previous accesses [68]. The RDF triples corresponding to the patterns that cross a specified frequency threshold are then prefetched into memory.

By far the most popular technique used to prefetch data in RDF Triplestores has been query augmentation. This technique, also called ‘query relaxation’, aims at relaxing the conditions of a query in order to prefetch additional data that is potentially needed for subsequent queries. While query augmentation predates RDF triplestores, it has been adapted many times to SPARQL queries as an approach to perform prefetching.

Using this technique, Hurtado *et al.* suggest relaxing SPARQL queries by logical relaxation of their triple patterns based on logical entailment and ontological metadata retrieved from the triplestore [42]. In contrast, Hogan *et al.* propose an approach to query augmentation that relies on precomputed similarity tables for attribute values [40] and evaluate different distance measures to calculate this similarity. Finally, Elbassuoni *et al.* utilize a language model derived from the knowledge base to perform query augmentation [28].

Given that these query augmentation techniques need data from the data source, they require at least some precomputations to be performed before they can be applied. Furthermore, they are not readily portable across triplestores since the required information might not always be available.

2.4.3 Computer Language-Based Prefetching

In the context of RDF Triplestores, computer language-based approaches are based on analyzing SPARQL queries in order to obtain relevant information in developing more accurate prefetching. Zhang *et al.* present such an approach by measuring similarity between SPARQL queries using a Graph Edit Distance (GED) function [84]. The authors then use previous queries similar to the current query to ‘suggest’ data for prefetching. A major drawback of this approach is that it only works if the exact same query is launched several times, making it more similar to caching than prefetching.

In the area of query augmentation, Lorey *et al.* propose an approach that measures the similarity between past SPARQL queries based on a bottom-up graph pattern matching algorithm [55]. These measurements are used to cluster similar queries together and create a *query template* for each cluster. The authors extend their work by combining the *query templates* with four different query augmentation strategies but do not reach any conclusive results on which strategy offers the best results [54].

2.4.4 Discussion

Our approach belongs to the last category of prefetching techniques, it is a computer language-based approach that analyzes SPARQL query logs to predict data for prefetching. It also applies query augmentation in order to prefetch the predicted data.

However, unlike previous approaches that fall into the same category (e.g. [55]), we do no directly launch an augmented query but use a two-step prediction process to predict the structure of the augmented query before individually predicting which triple patterns to use (as discussed in Chapter 5). This separation allows us to take the query structure into account without performing any graph matching between each pair of SPARQL queries.

When compared to approaches that use query augmentation based on data found in the triplestore, our approach is more flexible given that it does not require any specific information that might not be found in the store. Instead, we apply our query augmentation when needed using the query logs of the SPARQL endpoint.

Chapter 3

Static Code Analysis of Object-Oriented Applications

This chapter details the first contribution of this thesis **(C1)** by presenting an approach that uses static code analysis of object-oriented applications to predict access to data in Persistent Object Stores (POSs). The analysis is performed before the application is executed and hence does not cause any overhead. Moreover, the approach is fully automatic and does not require any manual input from the developer.

We start by introducing a motivating example that shows the limitations of current approaches to predict access to data in POSs, in Section 3.1, and the type of information we can retrieve by statically analyzing the source code of an OO application in Section 3.2. Afterwards, we formalize our proposed approach using the concept of type graphs in Section 3.3. We then implement the proposed static code analysis and discuss the implementation details in Section 3.4. Afterwards, we move on to study the viability of the implemented approach in Section 3.5 by answering the following research questions:

RQ1: What is the percentage of applications for which static code analysis can predict access to persistent objects?

RQ2: Can the proposed static code analysis be performed within a reasonable amount of time?

RQ3: What is the prediction accuracy of the proposed static code analysis?

RQ4: How much in advance can the proposed static code analysis predict access to persistent objects?

Finally, we summarize our findings and conclude this chapter in Section 3.6.

3.1 Motivating Example

Figure 3.1 shows the POS schema of a bank management system. In the figure, we can see various classes representing the entities of the system, such as *Transaction*, *Account* and *Customer*. Let's say that we want to update the customers of the accounts responsible for all the transactions to be in the name of the manager of the bank. However, as a security measure, the system restricts updates on accounts to customers of the same company as the customer currently owning the account.

In order to achieve this task, we need to retrieve and iterate through all the *Transaction* objects. We then navigate to the referenced *Account* and *Customer* until reaching the *Company* of each customer. Finally, we need to compare the company of the customer currently owning the account with the company of the bank manager.

The simplest prediction technique that can be applied in this case is the Referenced-Objects Predictor (ROP), as defined in Section 2.3. Applying ROP to our example means that, for instance, each time a *Transaction* object is accessed, the referenced *Transaction Type*, *Account* and *Employee* objects are predicted to be accessed along with it.

However, in order to accomplish our task we also need to access the *Customer* and *Company* objects which will not be prefetched. On the other hand, the *Transaction Type* and *Employee* objects will be prefetched with *Transaction* but in reality are not needed for the task at hand. To put this in numbers, if we have 100,000 *Transactions* the ROP would wrongfully predict access to as many as 200,000 objects in the worst case while missing another 200,000 objects that will be accessed.

The prediction accuracy of ROP can be improved by increasing the "fetch depth", i.e. the number of levels of referenced objects to predict. For instance, instead of only predicting access to *Transaction Type*, *Account* and *Employee*, which are directly

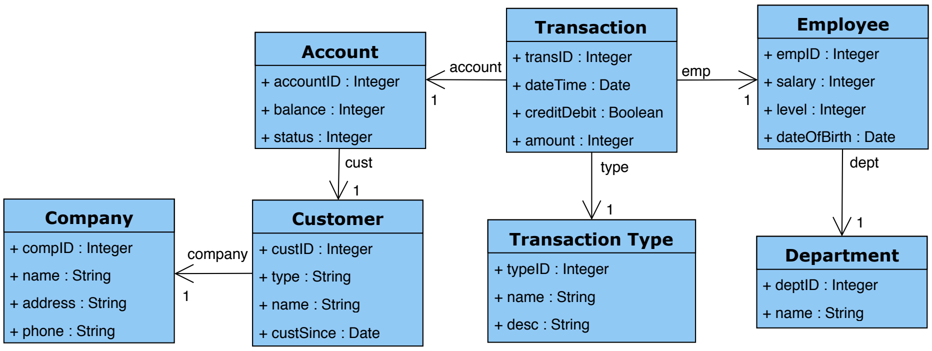


Figure 3.1: *Example of a Persistent Object Store (POS) schema. The schema represents a banking system with 7 entities, each of which corresponds to an object type in the POS.*

referenced from *Transaction*, having a fetch depth equal to 2 would also predict the objects referenced from them, which are *Department* and *Customer* in this example.

Increasing the fetch depth of ROP may help in predicting more relevant objects but it does not solve the problem of predicting access to objects that are not necessary. As a matter of fact, the more the fetch depth is increased the more likely it is to predict irrelevant objects as well. This is due to the fact that the ROP applies a heuristic based on the schema of the POS that does not take into account the application behavior.

Another more complex approach would be to monitor accesses to the POS and generate predictions based on the most commonly accessed objects [36, 43, 31]. For instance, monitoring accesses to the POS shown in Figure 3.1 might tell us that in 80% of the cases where a *Transaction* object is accessed, its related *Account* and *Customer* objects are accessed as well.

This would work perfectly for our task, we will only need to load the referenced *Company* object and all the other necessary objects will have been already prefetched. However, in the 20% of cases where a transaction's *Account* and *Customer* are not needed, they will still be prefetched despite the fact that they will not be accessed. Moreover, retrieving the necessary information for this approach requires runtime monitoring of the application which adds overhead to the application execution time [31].

The problem faced in both cases is that sometimes we prefetch objects that are not needed into memory and at the same time we don't prefetch objects that are actually accessed. This partially stems from the fact that the prediction heuristics are applied without taking into consideration the actual applications being used to access the data.

3.2 Solution: Static Code Analysis

Continuing with the same example, assume that we have an application with the partial implementation shown in Figure 3.2, written in an object-oriented language, to control access to the POS in Figure 3.1. The task that was described in Section 3.1 is implemented by the method *setAllTransCustomers()* (lines 30 to 34) in Figure 3.2. By analyzing the code of this method, we can see that whenever it is executed it accesses:

- the object *manager* defined in *BankManagement.*
- all of the *Transaction* objects defined in *BankManagement* by iterating through them in a *for* loop.

These objects are accessed directly by the method *setAllTransCustomers()* but the method also invokes other methods that may access persistent objects themselves. In particular, we see that *setAllTransCustomers()* invokes two methods:

- the method *getAccount()* which accesses the objects *Type*, *Employee* and *Account* referenced from a *Transaction.* Moreover, this method might also access the *Department* of the *Employee* of a *Transaction*, depending on which branch of the conditional statement starting on line 7 is executed.
- the method *setCustomer()* which accesses the objects *Customer* and *Company* referenced from the *Account* object returned by the invocation of *getAccount().*

By combining the information obtained from analyzing the method *setAllTransCustomers()* with that obtained from analyzing its invoked methods, we can get a better idea of which objects the method will access when executed. Performing this interprocedural analysis also permits to predict access to a persistent object with more time in advance. This is crucial to have sufficient time to prefetch the predicted objects before they are accessed by the application.

On the other side, performing a static analysis does not always give us certain information about which objects are accessed, such as the case with the method *getAccount()* which may or may not access the object *Department* depending on which branch of the

```
public class Transaction {
 private Account account;
 private Employee emp;
 private TransactionType type;

 public Account getAccount() {
  if (this.type.typeID == 1) {
   this.emp.doSmth();
  } else {
   this.emp.dept.doSmthElse();
  }
  return this.account;
 }
}

public class Account {
 private Customer cust;

 public void setCustomer(Customer newCust) {
   if (this.cust.company == newCust.company) {
     this.cust = newCust;
   }
 }
}

public class BankManagement {
 private ArrayList<Transaction> transactions;
 private Customer manager;

 public void setAllTransCustomers() {
   for (Transaction trans : this.transactions) {
     trans.getAccount().setCustomer(this.manager);
   }
 }
}

...
```

Figure 3.2: *Example application code written in an OOP language.*

conditional statement is taken. The branching behavior of an application is dynamically determined during runtime and cannot be fully studied statically. However, we demonstrate in the approach study in Section 3.5 that the branching behavior of an application has minimal effects on its access patterns to persistent objects.

Using all of this information obtained by statically analyzing the source code of an application, we can automatically generate method-specific access hints that predict which objects are going to be accessed by a method in the application. The clear advantage of this approach is that it is performed statically before the application is executed and hence does not add any overhead to the application execution time. The second advantage, which we show in Section 3.5, is that the prediction accuracy of this approach is higher than other approaches performed at compile-time.

3.3 Proposed Approach

The formalization of our approach is based on the notion of *application type graphs* presented in [43]. However, the approach in [43] predicts access to persistent objects by monitoring application execution while our approach performs the process by statically analyzing the application's code prior to its execution. To that end, we extend this formalism by introducing *method type graphs*, *branch-dependent navigations* and *augmented method type graphs*. We also formalize how *access hints* that predict access to persistent objects are obtained from a method's augmented type graph.

Let us assume that we have an object-oriented application that uses a POS with a one-to-one mapping between application objects and POS objects (i.e. each object in the application is exposed by the POS as an object regardless of how it is stored internally). Further assume that the application is already checked by a compiler and does not contain any compile-time errors.

We define T as the set of types (i.e. classes) and F as the set of fields of the application. Also, $\forall t \in T$ we define the two following sets:

F_t : the set of member fields of t

M_t : the set of member methods of t

3.3.1 Application Type Graph

An application type graph is a graph that captures the schema of the underlying POS by statically analyzing the source code of an application. Formally, the type graph of an application, as defined in [43], is a directed graph $G_T = (T, A)$ where:

- T is the set of types defined by the application.

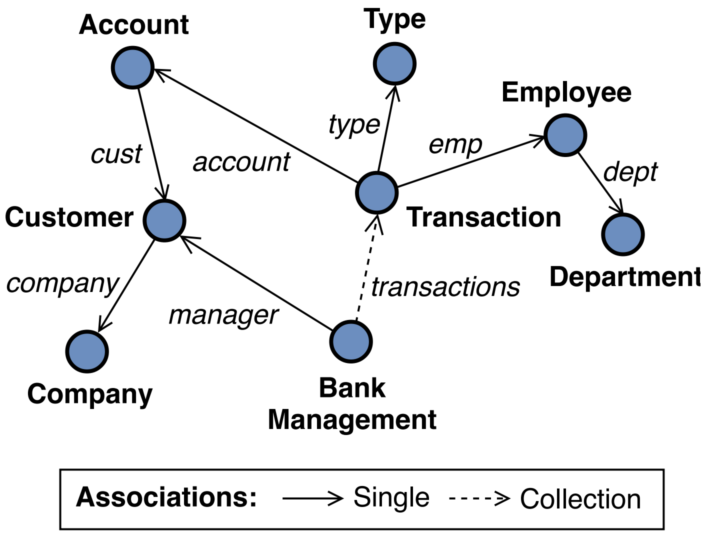


Figure 3.3: Type graph G_T of the application from Figure 3.1. Solid lines represent single associations and dashed lines represent collection associations.

- A is a function $T \times F \rightarrow T \times \{single, collection\}$ representing a set of associations between types. Given types t and t' and field f, if $A(t, f) \rightarrow (t', c)$ then there is an association from t to t' represented by $f \in F_t$ where $type(f) = t'$ with cardinality c, where c indicates whether the association is single-valued $(single)$ or multi-valued $(collection)$.

Example. Figure 3.3 shows the type graph of the application from Figure 3.1. There are two collection associations, namely *employees* and *transactions*. Some of the associations included in this type graph are formalized as:

- A(Employee, dept) ↦ (Department, single)
- A(Transaction, account) ↦ (Account, single)
- A(Bank Management, transactions) ↦ (Transaction, collection)

3.3.2 Method Type Graph

While the application type graph represents the general schema of the underlying POS, it does not capture how the associations between the different types are traversed by the methods of the application. In order to do so, we extend the formalism by defining a method type graph.

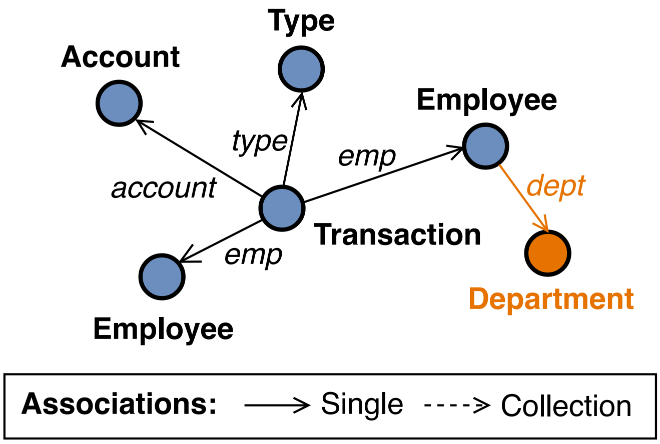


Figure 3.4: *Type graph G_m of the method getAccount() from Figure 3.2 (lines 6 to 13). Branch-dependent navigations (Section 3.3.3) are highlighted in orange.*

When a method m is executed, some of its instructions trigger the navigation of a subset of the associations in the application type graph. An **association navigation** $t \rightharpoondown^{f} t'$ is triggered by accessing a field f in an object of type t to navigate to an object of type t' such that $A(t, f) \rightarrow (t', c)$. We call the object of type t the *navigation source* and the object of type t' the *navigation target*. A navigation of a collection association has multiple target objects corresponding to the collection's elements. The set of all association navigations in a method m form a method type graph G_m, which is a sub-graph of G_T.

Example. Figure 3.4 shows the method type graph G_m of method *getAccount()* with the implementation shown in Figure 3.2 (lines 6 to 13). Notice that expressions such as *typeID* are not part of the graph because they do not trigger a navigation between objects (*typeID* is a field of a primitive type (integer) which is stored and retrieved automatically with its parent object).

3.3.3 Branch-Dependent Navigations

Since the execution of a method depends on the branching behavior of the program, not all of the association navigations are always triggered. For example, the navigation *Employee* $\rightharpoondown^{dept}$ *Department* from Figure 3.4 is triggered inside the *else* branch of the conditional statement starting on line 7 in Figure 3.2. This navigation might or might not be triggered during execution depending on which branch of the conditional statement is executed.

We call this type of navigations *branch-dependent navigations*, they are caused by the program's branching behavior, which depends on a method's conditional and loop statements. We identified two types of branch-dependent navigations:

- Navigations *not* triggered inside all the branches of a conditional statement. Since only one of the statement's branches is executed, a navigation that is not triggered in all of the statement's branches might not be triggered at all.

- Navigations of collection associations triggered in loop statements with branching instructions (i.e. *continue*, *break*, *return*) or increments greater than 1. The execution of such loops might be interrupted before all of the navigation's target objects (i.e. the elements of the target collection) are accessed.

Given that our analysis is done statically prior to the execution of the application, we cannot know which branch of the method will be executed, and thus which branch-dependent navigations will be triggered. To deal with the lack of this runtime information, we devised two different strategies to treat branch-dependent navigations; either include them in or exclude them from the method type graph G_m.

Including branch-dependent navigations in G_m might result in false positives if the branch from which the navigation is generated is not taken during execution. On the other hand, excluding them might result in a miss if the branch is indeed taken. In our approach study (Section 3.5), we evaluate both strategies and discuss which should be used in different circumstances.

Example. Looking at Figure 3.4 again, we can say the following about the navigations triggered inside the conditional statement starting on line 7 in Figure 3.2:

- $Employee \rightharpoondown^{dept} Department$ (highlighted in orange) is branch dependent since it is only triggered in one of the branches of the conditional statement.

- $Transaction \rightharpoondown^{emp} Employee$ is not branch-dependent (it is triggered inside both branches).

3.3.4 Augmented Method Type Graph

The method type graph, defined in Section 3.3.2, captures the association navigations directly triggered by the instructions of a method. However, a method m might also invoke other methods of the applications, which in turn trigger other association navigations. Since our goal is to predict access to persistent objects as early as possible, we also perform an inter-procedural analysis that links the method type graph of a method m with the type graphs of the methods invoked by m.

We do this analysis by constructing an augmented method type graph. An augmented method type graph $AG_m = (T_m \subseteq T, A_m)$ considers association navigations that cross the boundaries of a single method through method invocations. When a method m defined in type t invokes a second method m' defined in t', we construct AG_m by augmenting the type graph of m in two orthogonal ways:

- The type graph of the invoked method $G_{m'}$ is added to AG_m through the navigation $t \rightharpoondown^f t'$ that caused the invocation of m'.

- When a persistent object is passed as a parameter to m', the instructions of m' might also trigger navigations starting from the passed parameter and all such navigations are added to AG_m. Note that these navigations do not form part of $G_{m'}$ since parameters are considered local variables in their method's body.

Example. Figure 3.5 shows the augmented method type graph AG_m of method *setAllTransCustomers()* with the implementation shown in Figure 3.2 (lines 30 to 34). We can observe the following from this figure:

- $BankManagement \rightharpoondown^{transactions} Transaction \rightharpoondown^{transType} TransactionType$ mean that the navigation $Transaction \rightharpoondown^{transType} Transaction\ Type$ is triggered for each target object of the multi-valued association navigation $Bank\ Management \rightharpoondown^{transactions} Transaction$.

- *Account* $\rightharpoondown^{cust}$ *Customer* is the result of invoking the method *setCustomer(newCust)* on the return object of method *getAccount()*.

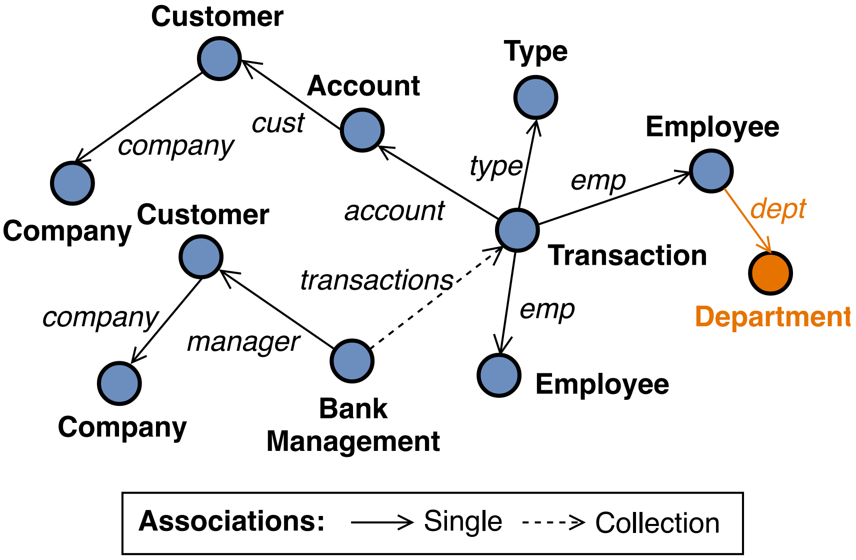


Figure 3.5: *Augmented type graph AG_m of the method setAllTransCustomers() from Figure 3.2 (lines 30 to 34). The navigations that cross method boundaries are caused by the invocations of methods getAccount() and setCustomer(newCust) on line 32 in Figure 3.2. Branch-dependent navigations (Section 3.3.3) are highlighted in orange.*

- $BankManagement \rightarrow^{manager} Customer \rightarrow^{company} Company$ are triggered by passing *manager* as a parameter to the method *setCustomer(newCust)*. They are independent of the navigation triggered to invoke the method since parameters are orthogonal to the method invocation object.

3.3.5 Access Hints

After constructing the augmented type graph of a method AG_m, we generate the set of access hints that predict access to persistent objects AH_m by traversing the graph. Each access hint ah maps to a unique path in the graph AG_m and indicates a sequence of accesses to persistent objects starting at the instruction triggering the first navigation in the path.

A path might contain navigations of both single and collection associations. In the case of a single association, the corresponding access hint indicates that a single object is accessed, namely the navigation's target object. On the other hand, a navigation of a collection association has multiple target objects and the corresponding hint in this case indicates that *all* of these objects are accessed.

Thus, we define the set of access hints of a method AH_m as:

$$AH_m = \{ah \mid ah = f_1.f_2.\ldots.f_n \;\; where \; A_m(t_i, f_i) \rightarrow (t_{i+1}, c) : 1 \leq i < n\} \qquad (3.1)$$

Example. The augmented method type graph AG_m of Figure 3.5 results in the following set of access hints for method *setAllTransCustomers()*. Note that hints starting with the collection *transactions* indicate that all its elements are accessed.

$$AH_m = \{transactions.transType,\; transactions.emp,$$
$$transactions.account.cust.company,\; manager.company\}$$

3.3.6 Overridden Methods

A main feature of OOP languages is type inheritance, which allows a type t' to be defined as a subclass of another type t. Inheritance allows applications to define an object of a type t but initialize it of any subtype of t. Moreover, any method m defined in t can have overridden versions, with different implementations, in any subtype of t. Thus, when a method is executed on an object defined of type t and initialized to a subtype, the application actually executes the overridden version of m.

This behavior is called Dynamic Binding, in which the type of an object and the methods being executed are not known until runtime [9]. Similar to branch-dependent navigations (defined in Section 3.3.3), this information cannot be retrieved by our static analysis, given that is it only available when the application is executed. Hence, we also devises different strategies to deal with this

Formally, if we have a type $t \in T$ that has one or more subtypes $ST_t \subset T$, we define the set of overridden versions of a method in that type $m \in M_t$ as:

$$OM_m = \{m' | m' \in t' \; where \; t' \in ST_t\} \qquad (3.2)$$

Due to Dynamic Binding, always using the access hints of m might lead to wrong predictions of which persistent objects should be prefetched when an overridden method of m is executed. In order to handle this case, we propose adding further access hints to AH_m using one of the following strategies, both of these strategies are evaluated and discussed in Section 3.5:

- $\bigcap_{m' \in OM_m} AH_{m'}$: adding the *intersection* of the access hints of all the overridden versions of m to AH_m.
- $\bigcup_{m' \in OM_m} AH_{m'}$: adding the *union* of the access hints of all the overridden versions of m to AH_m.

3.4 Approach Implementation

In order to validate our static code analysis approach, as defined in Section 3.3, we implemented it in Java using **IBM Wala** [82]. Wala is an open-source tool that uses Eclipse Java Development Tools (JDT) libraries to parse and manipulate Java source code. We chose to implement our approach on Java as it is the most common object-oriented language. However, the theoretical concepts of the approach can be applied to any other object-oriented language. We chose Wala as it was the most adequate static code analysis tool that contained the features we needed to implement our approach.

We used the Abstract Syntax Tree (AST) that Wala constructs in order to identify the conditional and loop statements of each method. We identified two loop patterns used to iterate collections: using indexes or using iterators, each of which can be implemented with a *for* or a *while* loop. We also took *if*, *if-else* and *switch-case* statements into consideration when identifying conditional statements.

Wala also generates an Intermediate Representation (IR) of Java source code that contains Java bytecode instructions along with a control flow graph, a symbol table and a local variable value-to-name mapping. We used this IR in our analysis in order to detect the instructions that trigger association navigations and construct the type graph and augmented type graph of each method of the application.

Figure 3.6 shows a class diagram of the implemented static code analysis approach. The diagram depicts both internal classes of Wala that were used in our implementation (in blue), as well as additional classes that we implemented to construct the type graphs and generate the access hints (in orange and green).

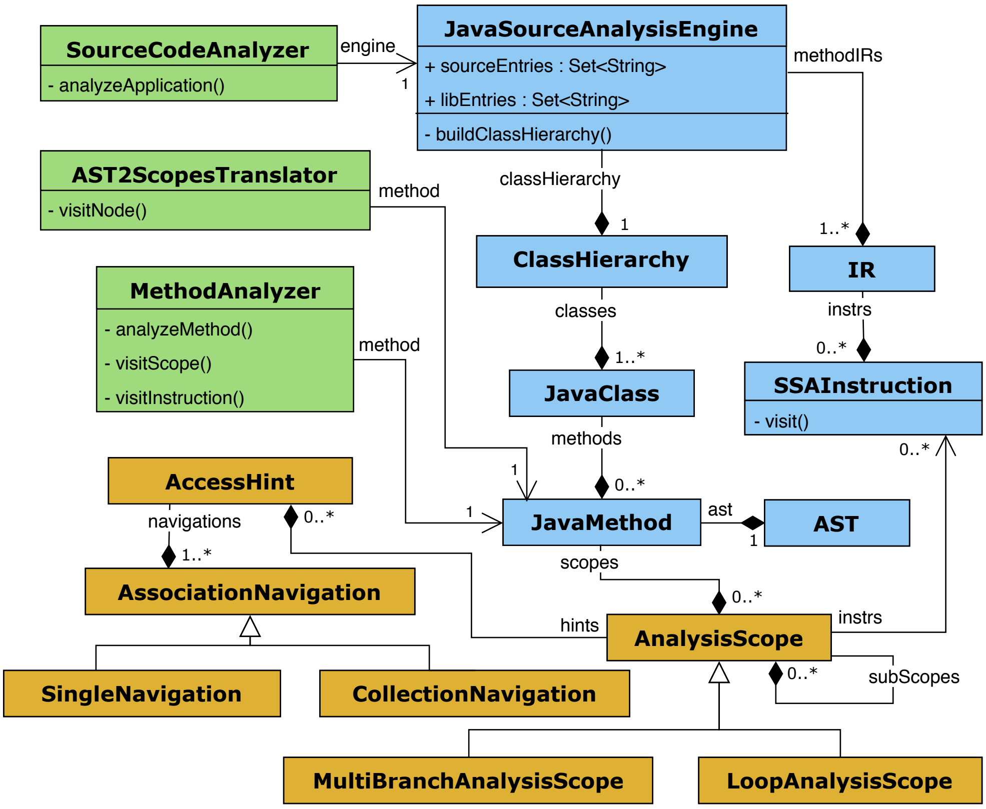


Figure 3.6: *Class diagram of the implemented static code analysis. Classes in blue represent internal Wala classes that are relevant to our analysis. Classes in orange represent additional data structures that we implemented to store the necessary data. Classes in green represent the main classes that we implemented to perform the analysis.*

3.4.1 Wala Abstract Syntax Tree

The analysis starts with the class *SourceCodeAnalyzer* that initializes Wala's *JavaSourceAnalysisEngine* with the set of source classes and required libraries of the application to be analyzed. Once it is initialized, this analysis engine builds a class hierarchy of the application classes and generates a *JavaClass* object for each class. It also generates a *JavaMethod* object for each method in the class as well as an Abstract Syntax Tree *(AST)* for each *JavaMethod*.

Example. Figure 3.7 depicts the AST of the method *setAllTransCustomers()* from Figure 3.2. We can see different types of nodes that represent the different instructions and statements from the method. For instance, the node *DECL_STMT* represents a variable declaration and has two child nodes; (1) the name of the defined variable and (2) the value which is assigned to it. In this case, a *DECL_STMT* node is used for the implicit declaration of the iterator of the collection *transactions* used in the for loop.

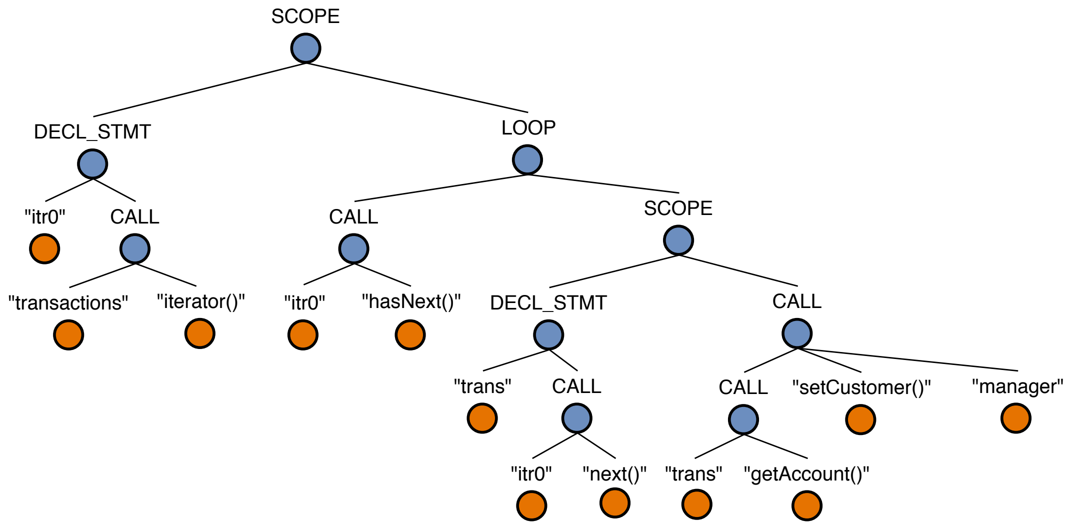


Figure 3.7: *Abstract Syntax Tree (AST) generated by Wala of the method setAllTransCustomers() from Figure 3.2. Nodes in blue represent the internal structure of the tree while nodes in orange represent the names of the variables and methods used in the instructions.*

Similarly, the node *CALL* is used to represent a method invocation and has at least two child nodes; the first represents the object on which the method is invoked and the second represents the invoked method. The *CALL* node can also have more children representing the parameters passed to the invoked method, such as the case with the node *"manager"*, which is a parameter of the invocation of the method *setCustomer()*.

3.4.2 Wala Intermediate Representation

Afterwards, the Wala analysis engine constructs the Intermediate Representation (IR) for each method and maintains a map of IRs separately from the *JavaMethod* objects. The IR objects contain a unique *value number* for each local variable defined in the method, including its parameters. For a non-static method, v_1 represents the *this* parameter. The parameters to the method are next (v_2, v_3, etc). For a static method, v_1 represents the first parameter. The local variables defined inside a method's body follow the parameters in numbering.

Moreover, the IR includes a set of *SSAInstructions* representing the bytecode instructions of the corresponding method. Each *SSAInstruction* consists of three parts:

- The instruction type (e.g. invocation, conditional branch) and the parameters of the instruction (e.g. the invoked method and its parameters),

- At most one variable defined by the instruction, which is given a new unique value number, and
- Zero or more variables that are already defined and used by the instruction, indicated by their corresponding value numbers.

Each *SSAInstruction* is also given an *Instruction Index (II)* which uniquely identifies it in the list of instructions inside the IR. Note that since the IR uses bytecode instructions as a basis, Wala might add additional instructions implicitly invoked in the original Java source code (such as converting loops to conditional branches and goto instructions).

Example. Figure 3.8 shows the IR instructions of the method *setAllTransCustomers()* from Figure 3.2. As explained above, the value v_1 represents the *this* parameter and given that the method does not have any declared parameters, the numbering of the values corresponding to the local variables of the method starts with v_2.

The line numbers shown in Figure 3.8 correspond to the index of each instruction inside the Wala IR. Note that the instructions $II_2, II_3, II_4, II_5, II_6$ and II_{10} are implicit instructions that Wala generates due to the for loop and are not explicitly invoked by the original method code. The instructions depicted in Figure 3.8 are the following:

- II_1: **getfield** instruction that loads the field *transactions*, of type *ArrayList*, defined in the class *BankManagement*. The instruction uses the value number v_1 (*this* parameter) to load the field and defines the value v_2.
- II_2: **invokemethod** instruction that invokes the method *iterator* defined in the class *ArrayList* on the value number v_2, which corresponds to the loaded *transactions* field defined by II_1. The returned iterator of the invoked method is assigned the value number v_3.
- II_3: **invokemethod** instruction that invokes the method *hasNext()* defined in the class *Iterator* on the value number v_3. The result of the invocation, a variable of type boolean, is assigned the value number v_4.

```
1  v2 = getfield < BankManagement, transactions, java/util/ArrayList > v1
2  v3 = invokemethod < java/util/ArrayList, iterator()java/util/Iterator > v2
3  v4 = invokemethod < java/util/Iterator, hasNext()B > v3
4  conditionalbranch (eq, to iindex = -1) v4, true
5  v5 = invokemethod < java/util/Iterator, next()java/lang/Object > v3
6  v6 = checkcast < Transaction > v5
7  v7 = invokemethod < Transaction, getAccount()Account > v6
8  v8 = getfield < BankManagement, manager, Customer > v1
9  invokemethod < Account, setCustomer(Customer)V > v7, v8
10 goto (from iindex = 10 to iindex = 3)
```

Figure 3.8: *Intermediate Representation (IR) generated by Wala of the instructions of the method setAllTransCustomers() from Figure 3.2. Line numbers represent the instruction indexes given by Wala. SSAInstruction names are in bold, invoked methods in blue and loaded class fields in red.*

- II_4: **conditionalbranch** instruction that checks if the variable v_4 equals the value *true* (i.e. if the *iterator* has more objects to load) and indicates the start of the loop statement in the original source code.

- II_5 an II_6: correspond to a method invocation that invoke the method *next* on the iterator defined by value number v_3 and check the type of the loaded object to correspond to the defined type of the original array list *Transaction*. If the type cast is successful, a new value number v_6 is given to the result of the operation, which corresponds to the explicitly defined local variable *trans*.

- II_7: an *invokemethod* instruction that performs the explicit invocation of the method *getAccount()* on the types object v_6. The returned object, of type *Account*, is given the value number v_7.

- II_8: a *getfield* instruction that loads the field *manager*, used as a parameter of the next instruction II_9, and assigns it the value number v_8.

- II_9: an *invokemethod* instruction that performs the explicit invocation of the method *setCustomer()*. This instruction uses two value numbers: v_7 corresponding to the object of type *Account* on which the method is invoked, and v_8 corresponding to the field *manager* used as a parameter of the invoked method.

- II_{10}: a **goto** instruction indicating the end of the loop statement and returning the execution to II_3.

3.4.3 Analysis Scopes

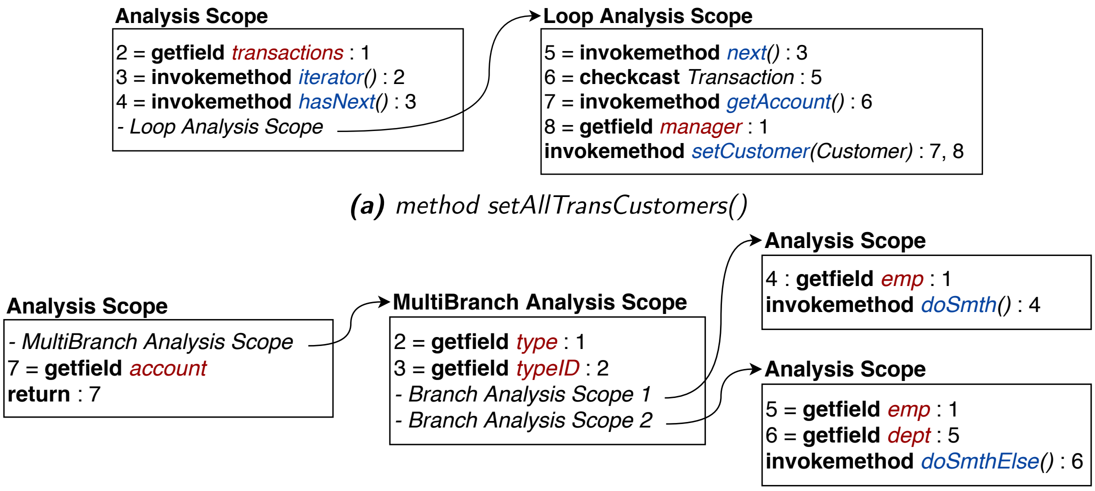
The internal Wala analysis engine keeps the AST and IR separate from each other. However, in order to perform our analysis, we need the AST created by Wala to detect conditional and loop statements and the IR to detect the instructions that trigger association navigations. Therefore, to facilitate the analysis, we construct *Analysis Scopes*.

The Analysis Scopes correspond to conditional and loop statements of the method and are used to identify branch-dependent navigations later on in the analysis. We create the scopes by implementing the visitor class *AST2ScopesTranslator*, which visits all the nodes in a method's AST and generates the corresponding scopes in a recursive manner. The visitor also stores the IR instructions generated by Wala inside the corresponding scopes before storing the scopes into the *JavaMethod* objects.

For each created analysis scope, we also identify the indexes of the first and last instruction inside the scope. These indexes are then used to more easily switch between the different scopes of a method when iterating through the method's instructions and performing our analysis, as detailed in Section 3.4.4.

Example. Figure 3.9 shows the analysis scopes constructed for two methods from Figure 3.2. In Figure 3.9 (a), we can see the analysis scopes of the method *setAllTransCustomers()* including the Loop Analysis Scope which corresponds to the *for* loop statement of the method. On the other hand, Figure 3.9 (b) depicts the analysis scopes of the method *getAccount()* and shows how the *if-else* conditional statement is represented by using one Multi-Branch Analysis Scope with two sub-scopes, each corresponding to one branch of the conditional statement.

Note that in both cases the IR instructions that are used in the conditions of the statements, namely the instruction `invokemethod hasNext()` in Figure 3.9 (a) and the instructions `getfield type` and `getfield typeID` in Figure 3.9 (b) belong to the parent scope of the scope representing the loop or conditional statement. We implemented this special case given that the instructions used to evaluate the conditions are always executed regardless of which branch is taken afterwards.


Analysis Scope
2 = getfield transactions : 1
3 = invokemethod iterator() : 2
4 = invokemethod hasNext() : 3
- Loop Analysis Scope
Loop Analysis Scope
5 = invokemethod next() : 3
6 = checkcast Transaction : 5
7 = invokemethod getAccount() : 6
8 = getfield manager : 1
invokemethod setCustomer(Customer) : 7, 8
Analysis Scope
- MultiBranch Analysis Scope
7 = getfield account
return : 7
MultiBranch Analysis Scope
2 = getfield type : 1
3 = getfield typeID : 2
- Branch Analysis Scope 1
- Branch Analysis Scope 2
Analysis Scope
4 : getfield emp : 1
invokemethod doSmth() : 4
Analysis Scope
5 = getfield emp : 1
6 = getfield dept : 5
invokemethod doSmthElse() : 6

(a) method setAllTransCustomers()

(b) method getAccount()

Figure 3.9: *Constructed Analysis Scopes of two methods from the application code in Figure 3.2. The instructions shown in this figure are a simplified representation of the Wala IR instructions.*

3.4.4 Method and Augmented Method Type Graphs

After creating the analysis scopes, the *SourceCodeAnalyzer* moves on to the main part of our analysis. It iterates through the engine's *JavaMethod* objects, now each containing the analysis scopes, and invokes another visitor class, *MethodAnalyzer*, for each method. *MethodAnalyzer* is the class responsible for visiting the scopes and instructions of a method and creating the method and augmented method type graphs. While these graphs are core concepts in our approach, they are never materialized or stored since they are intermediate structures used for the final goal of generating access hints.

Table 3.1 summarizes the IR *SSAInstructions* that were taken into consideration in our analysis. In order to detect single association navigations, we use the instruction *getfield* with the restriction that the type of the used field should be user-defined (i.e. the type should correspond to a class defined in the application). As for detecting collection association navigations, our analysis takes into account two types of instructions: **(1)** *arrayload* instructions, and **(2)** *invokemethod* instructions of the *next()* method of the *java.util.Iterator* class. A further restriction is also added to limit these navigations to instructions found inside loop analysis scopes.

Constructing the method type graph is then done by utilizing the used value numbers that Wala generates with the IR. For instance, when an instruction I_1 has in its

Table 3.1: *A summary of the Wala SSAInstructions used in our static code analysis approach.*

SSA Instruction	Restrictions	Used to Detect..
getfield	User-defined field type	Single association navigations
arrayload	Inside loop analysis scope	Collection association navigations
invokemethod	Inside loop analysis scope, method *java.util.Iterator.next()*	
checkcast	Inside loop analysis scope, User-defined cast type	
invokemethod	Method of user-defined class	Method invocations
return	*N/A*	Returned object of a method

used values the value number defined by I_2, the subgraph resulting from I_2 is linked to the node that represents the field accessed by I_1. In this case, I_2 can be any one of the instructions considered in the analysis as shown in Table 3.1.

Moreover, if the used value number corresponds to a method parameter, we mark it as such. This is important to bind the parameter to its value when constructing the augmented method type graph (see Section 3.3.4). We also take into consideration *return* instructions, if any, to detect the object that was returned by a method. The returned object is then used to link the result of a method invocation with further instructions (e.g. *getAccount().setCustomer()*).

In order to detect branch-dependent navigations, we considered the branching instructions *continue*, *break* and *return* when they occur inside a loop analysis scope. If such an instruction is detected, the navigations resulting from instructions inside the loop scope are marked as branch-dependent. On the other hand, all navigations resulting from the instructions of a child scope of a multi-branch analysis scope are marked as branch-dependent. These branch-dependent navigations are then dealt with later on during the generation of the access hints.

As for the augmented method type graph, it is constructed by detecting *invokemethod* instructions and augmenting the type graph of the method m in two orthogonal ways (as discussed in detail in Section 3.3.4): **(1)** The type graph of the invoked method is added to the type graph of m by linking it to the node representing the object that caused the invocation, and **(2)** The parameters of the invoked method are bound to the values used in the invocation and if these values correspond to persistent objects, their association navigations are added to the type graph of m.

These steps are detailed by the pseudo-code of Algorithm 1, which takes as input the source code of a method m and returns as output the augmented type graph AG_m. The algorithm iterates through the instructions of the method and creates new nodes in AG_m through the method *createNode()*, which takes as parameters the name of the node and whether it corresponds to a navigation of a single or collection association.

The method *createEdge()* is used to add an edge to AG_m between the node of the current instruction and the nodes of previous instructions, depending on the used and defined value numbers of the *SSAInstructions*. Finally, the algorithm switches the *currentScope* depending on the indexes of the first and last instruction in each scope of the method, in order to detect branch-dependent navigations when necessary.

Example. Applying the steps of Algorithm 1 on the instructions and analysis scopes of the method *setAllTransCustomers()* depicted in Figure 3.9 (a) results in the following (note that the instructions II_2, II_3 and II_5 do not access any persistent objects and hence do not cause changes to AG_m):

- The instruction II_1 = *getfield transactions* accesses a field of type *collection*. Hence, no changes are made to AG_m.

- II_4 is the first instruction of the loop analysis scope, hence the current scope is switched. The algorithm then detects that it is an invocation of the mthod *java.util.Iterator.next()*, which indicates that it is accessing elements of the collection *transactions* and a new node is added to AG_m.

- II_6 is an invocation of *getAccount()*. Hence, the type graph of *getAccount()* is added to AG_m. Similarly, II_7 results in adding the type graph of *setCustomer()* to AG_m, binding the method's parameter to the object *manager*.

Following this algorithm results in the construction of the augmented method type graph of *setAllTransCustomers()* depicted in Figure 3.5. An example of branch-dependent navigations is found in the analysis scopes of the method *getAccount()* depicted in Figure 3.9 (b), where we can see that the instructions *getfield emp* and *getfield dept* are found in child scopes of a Multi-Branch Analysis Scope. As such, the nodes and edges constructed from these instructions are marked as branch-dependent.

Algorithm 1: Construct Augmented Method Type Graph

```
Input   : $m \in M_t$: Source code of the method to analyze
Output: $AG_m$: Augmented Type Graph of the input method $m$
$AG_m \leftarrow (\phi, \phi)$
currentScope $\leftarrow \phi$
foreach $param \in P_m$ do
    $AG_m \leftarrow AG_m \cup$ createNode (param, isSingleOrCollection (param))
foreach $instr \in I_m$ do
    // Switch scopes depending on instruction index
    if $instrIndex$ = scopeStart $(m)$ then
        currentScope $\leftarrow$ getScope ($m$, instrIndex)
    else if $instrIndex$ = scopeEnd $(m)$ then
        currentScope $\leftarrow$ parent (currentScope)
    // Identify branch-dependent navigations
    if type (parent $(currentScope)$) = $multibranchanalysisscope$ ||
    (type $(instr) \in \{return, break, continue\}$ && $type(currentScope)$ =
     $loopanalysisscope$) then
        isBranchDependent $\leftarrow$ true
    // Handle SSA Instructions and create nodes in $AG_m$
    if type $(instr)$ = $getfield$ && fieldType $(instr) \in T$ then
        $AG_m \leftarrow AG_m \cup$ createNode (fieldName (instr), 'single')
    if ((type $(instr)$ = $arrayload$)
    || (type $(instr)$ = $invokemethod$ && invokedMethod $(instr)$ =
     $'java.util.Iterator.next()')$)
    && type $(currentScope)$ = $loopanalysisscope$ then
        $AG_m \leftarrow AG_m \cup$ createNode (usedValueName (instr), 'collection')
    if type $(instr)$ = $invokemethod$ && invokedMethod $(instr) \in M_T$ then
        $m' \leftarrow$ invokedMethod (instr)
        $AG_{m'} \leftarrow$ getMethodGraph (invokedMethod(instr))
        foreach $node \in AG_{m'}$ do
            if isParameterNode $(node)$ then
                replaceNode ($AG_{m'}$, node, bindParameter (node))
        $AG_m \leftarrow AG_m \cup AG_{m'}$
    if type $(instr)$ = $return$ then
        usedNode $\leftarrow$ getNode(usedValueName (instr))
        setIsReturnObject (usedNode)
    // Create edges between new nodes and used nodes
    definedNode $\leftarrow$ getNode (defValueName (instr))
    foreach $usedValueName \in$ usedValueNames $(instr)$ do
        usedNode $\leftarrow$ getNode (usedValueNumber)
        $AG_m \leftarrow AG_m \cup$ createEdge (usedNode, definedNode,
         isBranchDependent))
return $AG_m$
```

3.4.5 Access Hints

The final step in the analysis is also performed by the *MethodAnalyzer*. After creating a method's augmented type graph, as defined in Algorithm 1, the visitor generates *Access Hints*, each containing one or more *Association Navigations*, and stores them into the method being analyzed m. At this point, given that the nodes and edges of AG_m are marked to indicate branch-dependent navigations, we identify these navigations and either exclude or include them in the generation of the access hints, as discussed in Section 3.3.3. Finally, we apply one of the strategies defined in Section 3.3.6 to deal with overridden methods by supplementing the set of generated access hints of each method.

3.5 Approach Validation

To answer the research questions set out at the beginning of this chapter, we executed a set of experiments using the implementation of our approach explained in Section 3.4. We evaluated the implemented approach by analyzing the source code of the **SF110** corpus of applications. As proven in [29], SF110 is a statistically representative sample of 100 Java applications from SourceForge, a popular open source repository, extended with the 10 most popular applications from the same repository. Since the SF110 corpus is statistically representative, we believe that analyzing the source code of its applications gives us an accurate idea on the applicability of static code analysis of object-oriented applications to the generation of access hints.

We made several changes to the applications of the SF110 corpus for compatibility reasons with Wala. Table 3.2 reports these changes and indicates that four applications were excluded for the same motive, bringing the number of analyzed applications from the SF110 corpus to 106. Detailed information about the applications included in the SF110 corpus is available in [29].

For the sake of completeness, we supplemented the SF110 corpus with four additional applications. These applications include two benchmarks specifically designed for POSs and two benchmarks typically used for computation-intensive workloads:

Table 3.2: *The changes and exclusions we made in our experiments to the application from the SF110 corpus.*

Application	Changes / Exclusions	Justification
heal	Include javax.servlet source code instead of JAR file	Used JAR file not readable by Wala
javaviewcontrol		
jwbf	Use a newer version (3.0.0)	Old version contained compile errors
corina	Exclude method: *DecadalModel.setValueAt()*	*Do-while* statement inside a case of a *Switch-case* statement not supported by Wala
weka	Exclude methods: *LinearRegression.findBestModel()*, *GUIChooser.GUIChooser()*	
wheelwebtool	Exclude method: *JSONTokener.nextClean()*	
summa	Exclude methods: *SolrResponseBuilder.parseDoc()*, *SummonResponseBuilder.extractRecord()*	Anonymous classes not supported by Wala
sweethome3d	Exclude entire application	
vuze		
jcvi		Generic types where names are changed not supported by Wala
liferay		Java HeapSpace Exception is thrown by Wala while creating AST

- **OO7:** the *de facto* standard benchmark for POSs and object-oriented databases [17]. The benchmark's model is meant to be an abstraction of different CAD/-CAM/CASE applications and hence contains a recursive data structure involving many classes with complex inheritance and composition relationships.

- **JPAB** [1]**:** a benchmark that measures the performance of ORMs compliant with the Java Persistent API (JPA) using four types of workloads (persist, retrieve, query and update).

- **K-Means:** a clustering algorithm typically used as a big data benchmark.

- **Princeton Graph Algorithms (PGA)** [2]**:** a set of various graph algorithms with different types of graphs (undirected, directed, weighted). More specifically, we executed the Depth-First Search, Breadth-First Search, Dijkstra Shortest Path and Bellman-Ford Shortest Path algorithms.

[1] JPAB: http://www.jpab.org/Benchmark_FAQ.html

[2] PGA: http://algs4.cs.princeton.edu/40graphs/

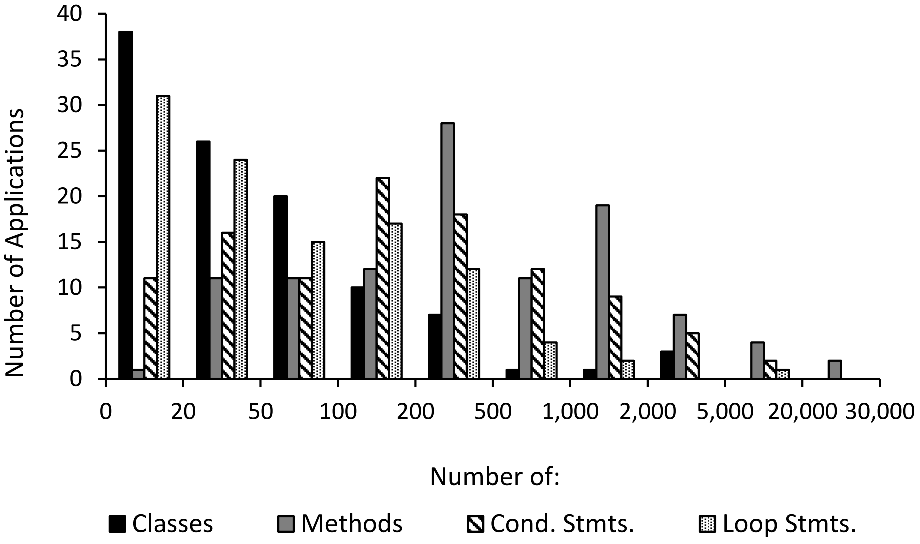


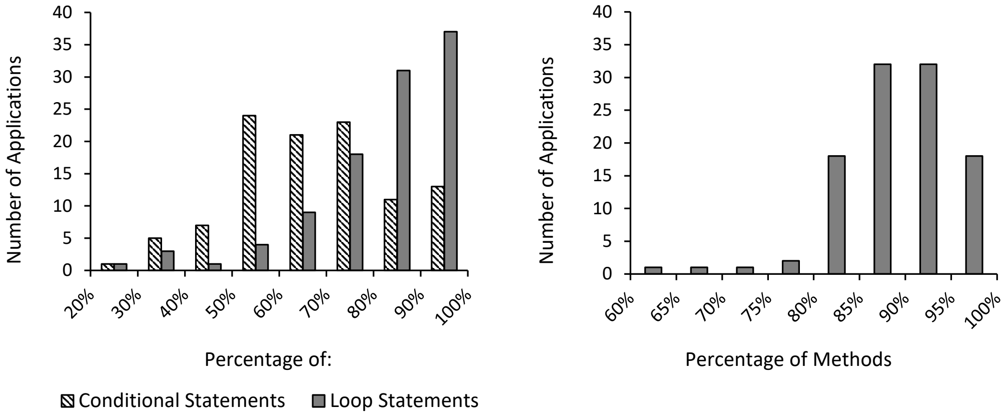
Figure 3.10: *For each interval, we report the number of applications used in our study that have the number of classes, methods, conditional statements and loop statements (as detected by our approach) in that interval.*

Table 3.3: *Summarized statistics of the corpus of applications used in our approach study.*

	Min	Max	Median	Avg	Std. Dev.	Total
# Classes	1	2,292	38	139	381	14,760
# Methods	2	26,261	335	1,379	3,517	146,182
# Cond. Stmts.	0	17,935	162	656	1,893	69,495
# Loop Stmts.	0	6,747	46	185	674	19,634

Figure 3.10 shows an aggregation of relevant characteristics of the applications used in our study: number of classes, methods, conditional statements and loop statements. Table 3.3 shows some summarized statistics of these characteristics and indicates that the test suite covers a wide range of applications, from very small applications to large applications containing over 20,000 methods.

The rest of this section explains the results obtained from our experiments and is divided into four parts, each answering one of the research questions we address. To answer **RQ1** and **RQ2**, we performed our experiments on the entire corpus of applications. Afterwards, we select a representative set of applications to use as benchmarks to answer **RQ3** and **RQ4**, which require runtime information that cannot be easily gathered for over 100 applications.



(a) *Conditional and Loop Statements* ***(b)*** *Methods*

Figure 3.11: *For each 5% or 10% interval, we report the number of applications that have the proportion of conditional statements, loop statements or methods that do not trigger any branch-dependent navigations (Section 3.3.3) in that interval.*

3.5.1 RQ1: What is the percentage of applications for which static code analysis can predict access to persistent objects?

To answer this question, we first analyzed the conditional and loop statements in the studied applications. Figure 3.11 (a) shows the number of applications per percentage of conditional and loop statements that do not trigger any branch-dependent navigations. The category axis of Figure 3.11 (a) starts at 20% as none of the analyzed applications scored less in either case. It should be noted that one of the studied applications, *greencow*, does not have any conditional statements while two, *greencow* and *dash-framework*, do not have any loop statements. Table 3.4 shows that an average of 67.5% of conditional statements and 82% of loop statements do not trigger branch-dependent navigations, and hence do not pose a problem when generating access hints.

We aggregated these results to calculate the percentage of methods of each application that do not trigger any branch-dependent navigations, i.e. the methods for which our approach predicts the exact set of persistent objects that will be accessed. Figure 3.11 (b) shows the results of this experiment, its category axis starts at 60% as only one of the studied applications, *jmca* [77], scored a lower percentage of 44.05%.

Figure 3.11 (b) shows that only 5 of the studied applications scored below 80%, which indicates that for 95.5% of the studied applications, our approach can generate

Table 3.4: *Summarized statistics of the experimental results. The first three rows show the percentage of conditional statements, loop statements and methods that do not trigger any branch-dependent navigations. The last row shows the analysis time of the studied applications.*

	Min	Max	Median	Avg	Std. Dev.
Cond. Stmts. (%)	26.8%	100%	67.1%	67.5%	17%
Loop Stmts. (%)	24.8%	100%	85.7%	82%	15.7%
Methods (%)	44%	100%	89.9%	88.8%	7.9%
Analysis Time (ms)	1	15,648	133	651	1,691

the exact set of access hints for over 80% of methods. Table 3.4 indicates that on average, 88.8% of an application's methods do not trigger branch-dependent navigations, which is significantly higher than the average reported for conditional and loop statements, and also reports a low standard deviation of 7.9%.

These results indicate that the prediction errors stemming from branch-dependent navigations are confined to a limited number of methods, while our static code analysis approach can accurately predict access to persistent objects in most cases. This is also in line with the intuition of the authors of [43] that accesses to persistent data are, in general, independent of an application's branching behavior.

3.5.2 RQ2: Can the proposed static code analysis be performed within a reasonable amount of time?

To answer this question, we measured the time that our proposed approach needed to analyze the studied applications. Figure 3.12 plots the number of applications per range of analysis time in milliseconds and shows that our approach only needs more than 2 seconds for 6 of the studied applications. Table 3.4 reports an average analysis time of 651 milliseconds and a maximum of just over 15 seconds which was measured when analyzing *weka*, the second largest application with over 20,000 methods.

As expected, the analysis time of our approach is correlated with the number of classes and methods of an application. However, with an average analysis time of 651 milliseconds and a maximum of roughly 15 seconds, we believe that the analysis finishes within a reasonable time for all of the analyzed applications. It is worth mentioning again here that this static analysis is done prior to application execution and does not add any overhead to its execution time.

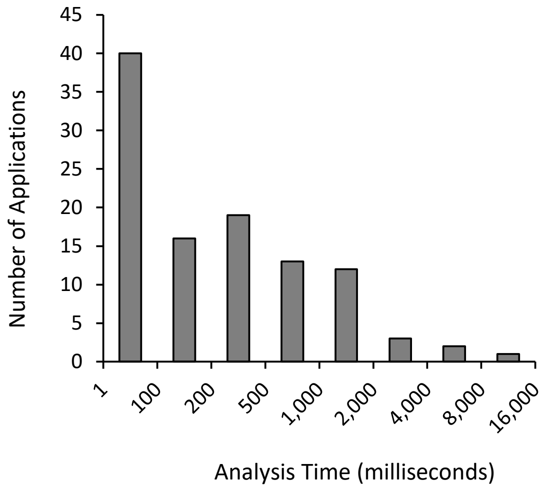


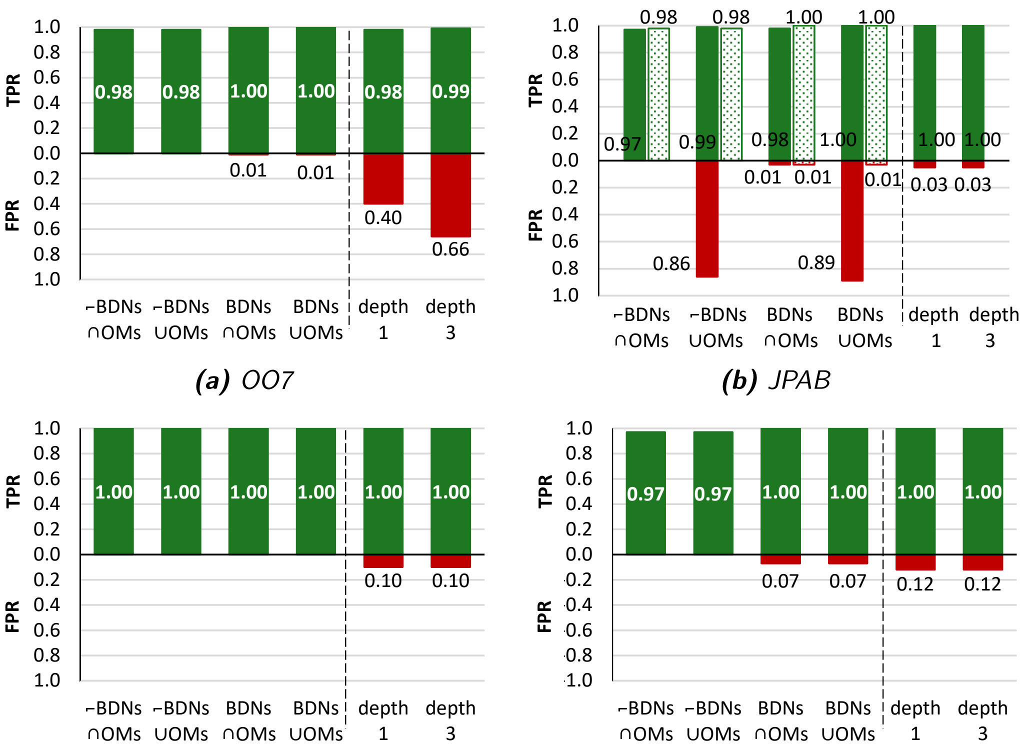
Figure 3.12: *For each time interval, we report the number of applications for which our approach finishes within that interval.*

3.5.3 RQ3: What is the prediction accuracy of the proposed static code analysis?

To answer this question, we tested the different strategies proposed in Section 3.3 to deal with branch-dependent navigations and overridden methods. We ran this experiment with the four applications that we added to the SF110 corpus; OO7, JPAB, K-Means and PGA, described in Section 3.5. We also compared our approach with the ROP with fetch depths of 1 and 3 in order to test its effect on the results. We used Hibernate 4.1.0 with PostgreSQL 9.3 as the persistent storage in all the experiments.

Figure 3.13 shows the True Positive Ratio (correctly predicted objects / accessed objects) and False Positive Ratio (incorrectly predicted objects / total predicted objects) of these strategies compared with the *Referenced-Objects Predictor (ROP)*. We can see from Figure 3.13 that increasing the ROP's depth from 1 to 3 only affected the results of *OO7*, where it resulted in an increase in true as well as false positives. Figure 3.13 also shows that regardless of the used strategy, our approach results in fewer false positives than the ROP in all of the studied applications.

The only exception is taking the union of overridden methods' access hints with *JPAB*, represented by the solid-colored set of columns in Figure 3.13 (b), which results in a dramatic increase in false positives. This is due to the implementation of *JPAB*



(a) OO7 *(b) JPAB* *(c) K-Means* *(d) Princeton Graph Algorithms*

Figure 3.13: *True Positive Ratio (TPR) and False Positive Ratio (FPR) of our approach (left of the dashed line) compared with the ROP with a depth of 1 and 3 (right of the dashed line). Columns represent the following:*
- $\neg BDNs \cap OMs$: *excl. branch-dependent navigations / inter. of overridden methods*
- $\neg BDNs \cup OMs$: *excl. branch-dependent navigations / union of overridden methods*
- $BDNs \cap OMs$: *incl. branch-dependent navigations / inter. of overridden methods*
- $BDNs \cup OMs$: *incl. branch-dependent navigations / union of overridden methods*

which includes five different tests each with its independent set of model classes, all of which are subclasses of a common abstract class called *TestEntity*. Hence, taking the union of overridden methods' access hints results in predicting access to many objects unrelated to the test being executed. We did not face this problem with other applications that include inheritance, such as *OO7*, since this case is different from the usual use of inheritance where subclasses represent subtypes of the parent class and have more resemblance among each other.

To test the effect this specific case has on the results of *JPAB*, we reran the analysis excluding the methods of the abstract class *TestEntity* and their overridden versions. The results are shown by the dotted set of columns in Figure 3.13 (b) and indicate that excluding this case, *JPAB* exhibits the same behavior as the other studied applications.

Based on the results of this experiment, we draw the following conclusions:

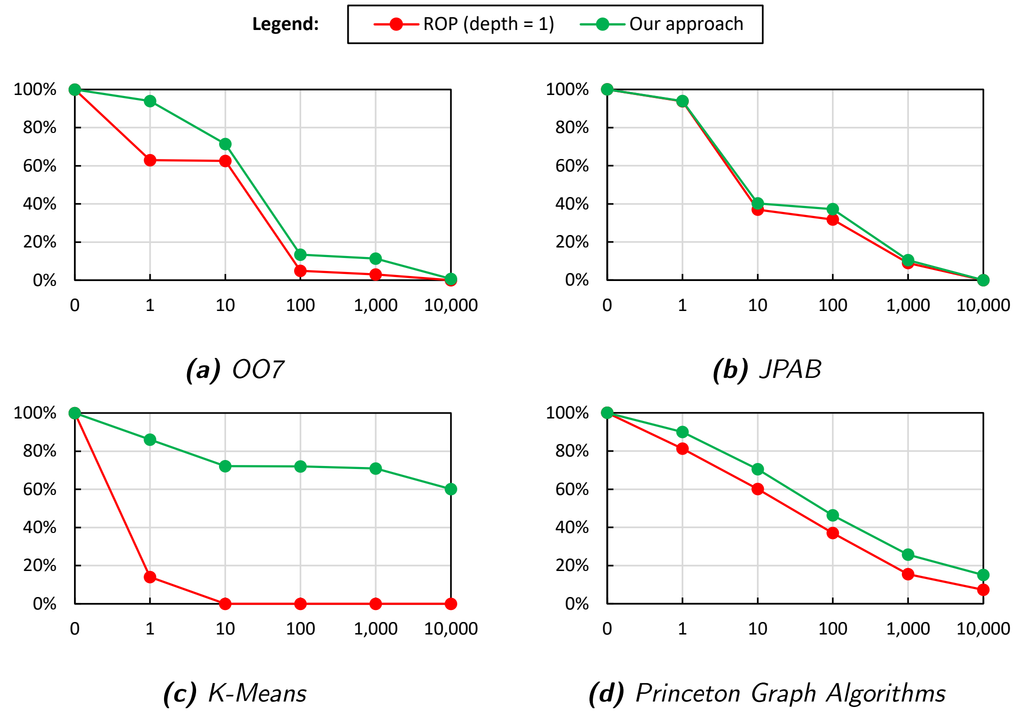
- Excluding branch-dependent navigations does not result in any false positives but causes a slight decrease in the true positive ratio when compared with ROP. This strategy should be used when memory resources are scarce since its predictions are accurate and will not occupy any memory with unnecessary objects.

- Including branch-dependent navigations might result in more false positives in some cases, such as with *OO7* and *Princeton Graph Algorithms*. However, it still generates fewer false positives than the ROP and the same true positive ratio, except with *JPAB* where achieving the same true positive ratio requires us to include the union of overridden methods' access hints as well. Hence, this strategy should be used when we are willing to sacrifice some memory, which will be occupied with false positives, in return of a higher true positive ratio.

- The strategy used to deal with overridden methods only affected the results of *JPAB*, where taking the union resulted in a sharp increase in false positives. As explained earlier, this is a specific case which can be easily identified and isolated but in general, we recommend to always take the intersection of overridden methods' access hints in order to avoid similar problems.

3.5.4 RQ4: How much in advance can the proposed static code analysis predict access to persistent objects?

In order to test how much in advance we can predict an access to a persistent object, we calculated the *distance*, measured by number of persistent accesses, between the time that an object o is predicted to be accessed and the actual access to o. Figure 3.14 shows the results of this experiment mapping each *distance* on the x-axis with the percentage of accesses that are predicted for that distance. For example, Figure 3.14 shows that with *OO7*, 95% of predictions made by our approach are done at least 1 persistent access in advance and 70% at least 10 persistent accesses in advance. We only compared our approach with the ROP with a depth of 1 since using depth 3 did not change the results of the previous experiment for most applications.



(a) *OO7*

(b) *JPAB*

(c) *K-Means*

(d) *Princeton Graph Algorithms*

Figure 3.14: *The x-axis represents the distance, in number of persistent accesses, between the prediction of an access to a persistent object o and the actual access to o. The y-axis represents the percentage of accesses that are predicted for each distance.*

Figure 3.14 shows that in the case of *JPAB*, the improvement we obtain over the ROP is very small because the application's data model and code structure do not allow for predictions to be made far in advance. However, in the case of *OO7* our approach can predict over 30% more accesses than the ROP with a *distance* of 1. Moreover, the significant improvement in the case of *K-Means* is due to the fact that it only has two persistent classes and relying on the relations between these classes does not allow the ROP to predict any accesses beyond a distance of 10.

This experiment indicates that another advantage of our approach is that it can predict accesses to persistent objects with more time in advance than the ROP. This indicates that the predictions are known with enough time in advance for the predicted objects to be prefetched *before* they are accessed by the application.

3.6 Summary

In this chapter, we presented our first contribution **(C1)**; an approach to predict access to data in Persistent Object Stores based on static code analysis of object-oriented applications. The symmetry between application objects and persistent objects provides the ideal setting for our approach as information about the persistent objects can be gathered by studying the application code. We formalized the approach using the concept of type graphs and explained how we use these graphs to generate *access hints* that predict which persistent objects are accessed by an application.

In our approach validation, we ran a series of experiments that demonstrate the viability of our approach. The experimental results show that the proposed static code analysis can generate exact access hints for the majority of studied applications and that problems stemming from the lack of runtime information only have minor effects. When compared with the *Referenced-Objects Predictor*, we have shown that our approach offers higher-accuracy prediction and makes the predictions with more time in advance.

We conclude from this chapter that statically analyzing the program code of object-oriented applications can indeed be used to generate accurate, compile-time predictions on which persistent objects are accessed by the application. These predictions can then be used for a variety of techniques that aim to improve access times to data, such as prefetching, cache replacement policies and dynamic data placement. For the purposes of this thesis, we take advantage of the predictions to develop an approach to prefetching for Persistent Object Stores, as explained in Chapter 4.

Chapter 4

Prefetching in Persistent Object Stores

This chapter presents the second contribution of this thesis **(C2)**, which consists of a prefetching technique for Persistent Object Stores based on the analysis of application code written in object-oriented languages. In this contribution we integrated the static code analysis approach discussed in Chapter 3 into *dataClay*, a POS currently being developed at the Barcelona Supercomputing Center.

dataClay is a distributed POS designed to share data with external players in a secure and flexible way based on the concepts of identity and encapsulation [60]. In contrast with other database systems, data stored in *dataClay* never moves outside the POS. Instead, every single piece of data can be uniquely identified and individually manipulated by the set of operations that encapsulate it, which are executed inside the data store. That is, data is manipulated in the form of objects, exposing only the operations that can be executed on the data, instead of exposing the data itself [60].

We used *dataClay* as an example POS into which we integrated our prefetching approach due to various reasons. Firstly, *dataClay* does not only store the data but also the operations and code that is used to access this data. Given that our approach is based on static code analysis, we believed that this feature would be helpful in facilitating the integration process. Moreover, *dataClay* is an open source project which allows us to easily modify the code and add any features needed for our integration.

While *dataClay* itself is not a contribution of this thesis, we modified its source code to accommodate the integration of our prefetching approach. In this chapter, we first offer an overview of the main features of *dataClay* that were used in the integration of our prefetching approach in Section 4.1. Afterwards, we discuss the implementation details of the static code analysis module in Section 4.2 and describe how the generated access hints are used at runtime to prefetch objects within *dataClay* in Section 4.3. We then move on to discuss our evaluation and experimental results in Section 4.4 by answering the following research questions:

RQ5: Does our prefetching approach improve application execution times?

RQ6: What is the object hit rate of the prefetching approach?

Finally, we summarize our findings and conclude the chapter in Section 4.5.

4.1 dataClay Overview

Figure 4.1 shows the system architecture of dataClay and communications between the user and *dataClay*. The figure depicts three main parts of the dataClay system:

- **dataClayTool:** a user-level application that facilitates management operations related with data access control (e.g. granting third-party users access to data) and the definition and sharing of application schema.
- **Logic Module:** a centralized service that receives all requests resulting from the *dataClayTool*. It also contains a central repository of object metadata, which is distributed by aggressive caching in all of the Data Services (see next point) and updated periodically.
- **Data Services:** one or more nodes where the objects are actually stored by *dataClay*. They handle all object operations including persistence requests (e.g. store, load, update) and execution requests. These Data Services communicate both with the Logic Module, in order to send or retrieve missing metadata, as well as with each other, in order to propagate requests received from the client.

This figure is not meant to be a comprehensive depiction of all the functionality that *dataClay* offers but rather show the main features of *dataClay* that are relevant to this thesis. In the remainder of this section, we explain each of these features.

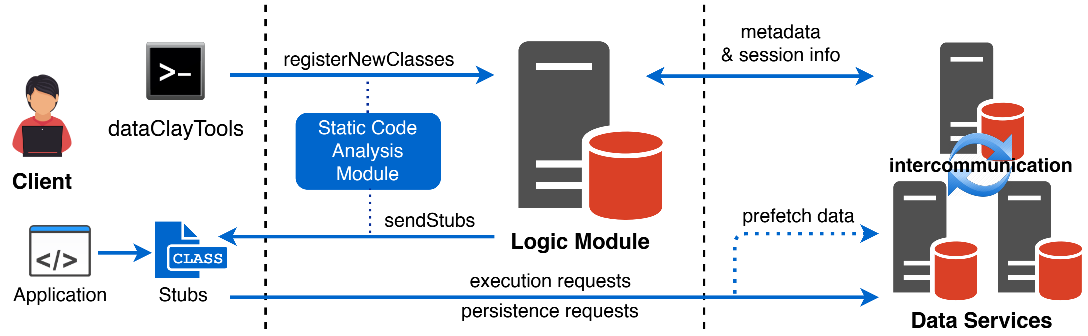


Figure 4.1: *Overview of the system architecture of dataClay. In this example, a deployment of a Logic Module and three Data Services on different nodes is depicted. Communications between the client and dataClay and intercommunication between the Logic Module and Data Services are depicted as well [60].*

4.1.1 Schema Registration

The first step in using *dataClay* involves the client registering the schema of his or her application in the system, i.e. the set of classes that will by used by the application, which is done through *dataClayTool*. When the client sends the classes to *dataClay* for registration, *dataClayTool* analyzes the source code to verify that all class dependencies are found in the given paths and automatically registers all the required classes transparently to the client. Once the classes are received by the Logic Module, it deploys them to the *Data Services* and the classes become ready to use.

After the classes are registered and deployed, the Logic Module sends a set of *stub classes*, one stub for each registered class, to the client. These stubs implement the functionality of the original registered classes and also extend from a global class called *DataClayObject*, which adds functionality related to object persistence and management. The client application that uses *dataClay* should then be compiled using these stubs instead of the original classes.

Our contribution to *dataClay* comes during this registrationg process. Once the classes are registered by the Logic Module and before the stubs are sent to the client, we implemented a **Static Code Analysis Module** that performs our analysis approach, as described in Section 3.3, and generates all the access hints necessary for the registered classes. Furthermore, the module injects code into the stubs generated by the Logic Module to add support for prefetching. Section 4.2 describes the implementation and integration details of this Static Code Analysis Module.

4.1.2 Data Generation and Persistence

After the application schema is registered, the client can proceed with creating data and storing it into *dataClay*. Data is stored in *dataClay* by creating a local object with the type of a registered class and then launching a **persistence request** on the created object. At this point, the client can specify an alias for the object which is then used to retrieve it. Furthermore, the client can also specify which Data Service they want the object to be stored in. If no particular Data Service is specified, a Data Service is chosen randomly using a hash function [60].

When *dataClay* is deployed with one Data Service, all objects are evidently stored in that Data Service. However, when several Data Services are deployed, the used hash function guarantees that the objects are distributed among the different Data Services. This automatic distribution of objects in *dataClay* is manged by the stub classes generated by the Logic Module and completely transparent to the client.

4.1.3 Remote Execution

After the objects are stored in *dataClay*, the client retrieves them either by using the alias assigned to them with the persistence request or by traversing the references between the objects. After retrieving an object, the client can execute any method from the object's stub class. At this point, the class stubs sent to the client after the registration process behave as Remote Procedure Calls (RPCs) for the object. Instead of executing the method locally, they launch an **execution request** that is submitted to the Data Service where the object is located [60].

When the execution request is received by the Data Service, the Data Service loads the object into its local memory and executes the method sent with the execution request. Thus as we can see, objects in *dataClay* are automatically loaded when needed, i.e. when an execution request on the object is received or an object is accessed through its alias or through a reference, and the client does not need to load them manually.

If during the method execution another object located in a different Data Service is needed, the Data Service does not load the object locally but rather generates a new execution request and sends it to the Data Service where the object is located, much like the application did in the first place. The motivation behind this behavior is to ensure that the methods are executed in the same Data Service where their objects are located, thus minimizing the movement of objects between different Data Services.

Given the changes made by the Static Analysis Module during the registration process (see Section 4.2), the access hints of the requested method are triggered once an execution request is received by a Data Service of *dataClay* in order to start prefetching the necessary data. Section 4.3 provides a detailed description of the steps followed in order to perform this data prefetching at runtime.

4.1.4 Lazy Tasks

dataClay also has a feature that allows methods to be scheduled for later execution by the Data Services. These methods are called *lazy tasks* and can be any method from any registered class in *dataClay*. The lazy tasks are handled by *lazy task runners* and each Data Service has its own lazy task runner, which is a timer thread with an associated queue of tasks.

A Data Service schedules a lazy task for execution by inserting it into its lazy task runner's queue. The lazy task runner then periodically checks the queue and when it finds a queued task, it fetches it for execution in its own thread separate from the main execution thread. External clients of *dataClay* cannot schedule lazy tasks since this feature is only used internally by *dataClay* for management operations. In our case, we take advantage of this feature in order to perform prefetching without interrupting the application execution, as will be discussed in detail in Section 4.3.

4.2 Static Code Analysis Module

The *Static Code Analysis Module* is our first contribution to dataClay. It implements our static code analysis approach, as described in Section 3.4, and prepares the stubs sent to the client for prefetching by injecting various pieces of code into them.

Figure 4.2 shows the workflow of the static code analysis module. When a client registers a new set of classes in *dataClay*, the module intercepts the call and performs the analysis. The first step is to use Wala to generate an Abstract Syntax Tree (AST)

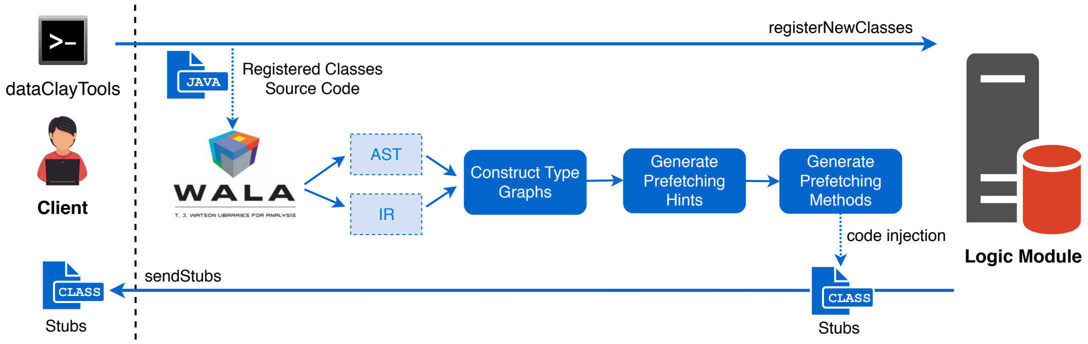


Figure 4.2: *A detailed view of the Static Code Analysis Module of dataClay.*

and Intermediate Representation (IR) of the source code of each method in the classes. Afterwards, we use these two structures to construct the type graphs and generate the access hints for each method, as defined in Section 3.3. Finally, we generate helper methods that prefetch the objects identified by the access hints and inject them into the corresponding class stubs before they are sent to the client.

4.2.1 Generating Type Graphs and Access Hints

We integrated our implemented approach into *dataClay* in order to construct the type graphs and generate the access hints of the registered applications. The details of this implementation are found in Section 3.4, the only change that we made in the approach is to filter access hints already found in previous method calls.

In Section 3.3, we explained how we perform inter-procedural analysis in order to generate access hints that predict access to objects found in several methods at once. By doing so, a method m that invokes another method m' will have the access hints resulting from the analysis of both m and m'. This allows to bring the prefetching forward ensuring that the predicted objects are prefetched before they are accessed. However, it also means that m and m' might have an access hint predicting access to the same object. On a theoretical level this does not pose a problem since the second access hint in m' simply predicts access to an object that is already prefetched in m.

However, when integrating our approach into *dataClay*, launching several requests to prefetch the same object might cause additional unnecessary overhead. We solved this problem with the following solution: for each method m, after generating its set

```
public void setAllTransCustomers_prefetch() {
   for (Transaction trans : this.transactions) {
     trans.type.load();
     trans.emp.load();
     trans.account.cust.company.load();
   }
   this.manager.company.load();
}
```

Figure 4.3: *Generated prefetching method for the method setAllTransCustomers() from Figure 3.2.*

of access hints AH_m, we removed from AH_m those access hints that are found in *all* of the methods that invoke m. This solution has no effect on the accuracy of our approach since the access hints removed from m are found in its invoking methods. Moreover, it is guaranteed that the objects predicted by the removed access hints will be loaded by other access hints in a previously executed method.

4.2.2 Generating Prefetching Methods

Once the access hints are generated, we generate, for each method, a helper prefetching method that contains instructions that load the objects corresponding to its access hints. We then use AspectJ to inject the generated prefetching method into the stub of the original method's class. Note that since these access hints result from the augmented type graph of the method, as defined in Section 3.3.4, the methods' instructions also load objects accessed by methods further down the callstack.

Example. For instance, the method *setAllTransCustomers()* from Figure 3.2 has the following set of access hints, as defined in Section 3.3.5:

$$\begin{aligned} AH_m = \{&transactions.transType,\ transactions.emp, \\ &transactions.account.cust.company,\ manager.company\} \end{aligned}$$

Thus, the corresponding prefetching method *setAllTransCustomers_prefetch()* that loads the objects defined by these access hints has the implementation shown in Figure 4.3. Note that we iterate through the elements of the collection *transactions* in order to load all of them.

Since *dataClay* loads objects automatically whenever an execution request on the object is received by the Data Service, the objects in the access hints are loaded when the next object is accessed. For example in the last access hint on line 7, the object *manager* is automatically loaded by *dataClay* when the access to its related object *company* is requested.

However, in order to load the last object in an access hint, we also injected an empty method called *load* into the class stubs of the registered classes in order to guarantee that the last object of each access hint is also loaded. The use of this method can be seen in the example prefetching method shown in Figure 4.3.

4.2.3 Invoking Generated Prefetching Methods

After the prefetching methods are generated, we need to modify the original application code to invoke these methods. However, instead of using a direct invocation we take a different approach that's less intrusive to the application. Since prefetching is meant to be a performance improvement, and since the prefetched objects are not all immediately needed, the overhead caused by prefetching should be minimized. To this end, we opted to use a multi-threaded approach where the prefetching methods are executed by a background thread in parallel to the main thread executing the original application code. By doing so, we allow the execution of the application to continue uninterrupted while at the same time prefetch objects whenever possible.

Hence, instead of injecting a direct invocation of the prefetching method, we use the *lazy tasks* feature of *dataClay* to inject an instruction that adds the prefetching method to the lazy task runner's queue. The lazy task runner will then only execute the prefetching method when the main thread of the application is not accessing the disk and hence minimize the prefetching overhead.

Example. Continuing with the same example, we modify the code of the method *setAllTransCustomers()* in the stub generated by the Logic Module to include the scheduled invocation of the prefetching method *setAllTransCustomers_prefetch()* as shown in Figure 4.4.

```
1 public void setAllTransCustomers() {
2   // Injected scheduling of prefetching lazy task
3   dataService.addLazyTask(setAllTransCustomers_prefetch());
4
5   for (Transaction trans : this.transactions) {
6     trans.getAccount().setCustomer(this.manager);
7   }
8 }
```

Figure 4.4: *Modified code of the method setAllTransCustomers() in the stubs generated by dataClay. Notice the injected call to add a lazy task with the prefetching method on line 3.*

4.2.4 Parallelization of Prefetching Methods

The prefetching methods described in the previous section help in improving the access times of an application by loading objects before they are needed. When the application uses a single Data Service where all the objects are stored, nothing more can be done to minimize the execution time of the prefetching methods. However, in the case of applications running on several Data Services, the methods do not take full advantage of the data distribution of objects.

For instance, in the prefetching method shown in Figure 4.3, the elements of the *transactions* collection and their related objects are loaded sequentially even though they might be located in different Data Services. A more efficient approach would be to load the elements in parallel in order to take advantage of the automatic data distribution of *dataClay*. On the other hand, distributing single-association hints, such as *manager.company*, is not possible since we need to guarantee that the object *manager* is loaded before we load its associated *company* object.

Hence, our implementation also includes **automatic parallelization of all possible prefetching instructions** in order to take full advantage of the automatic data distribution in *dataClay*. We implemented this feature by using the Parallel Streams of Java 8, which convert a collection into a stream and divide the stream into several substreams. The Java Virtual Machine (JVM) then uses a predefined pool of threads, namely the ForkJoinPool, to execute a specified task for each substream in parallel.

This is a very efficient way to do multithreading since the tasks are distributed among already-existing threads, hence avoiding the costs of creating and destroying

```
public void setAllTransCustomers_prefetch() {
   this.transactions.parallelStream().forEach(trans -> {
     trans.type.load();
     trans.emp.load();
     trans.account.cust.company.load();
   });

   // Cannot be parallelized
   manager.company.load();
}
```

Figure 4.5: *Automatic parallelization of prefetching method setAllTransCustomers_prefetch().*

threads for each task. Moreover, the number of threads in the ForkJoinPool is set by JVM to the number of processor cores of the current machine and the managemenet of the threads is done automatically by the JVM.

In our case, since we use the different threads to load different objects from disk, we only see a benefit when multiple Data Services are used. In the case of using one Data Service, the threads execute in parallel but the bottleneck remains the access to disk and hence parallelization does not produce any benefit. An example of this parallelization approach is depicted in Figure 4.5, which shows the parallel implementation of the prefetching method *setAllTransCustomers_prefetch()*.

Complete Example

Figure 4.6 shows all the injections we made into the class stub of the class *BankManagement*. We can see the injected *load()* method, the injected parallel prefetching method *setAllTransCustomers_prefetch()* as well as the injected scheduling of the prefetching method inside the method *setAllTransCustomers()*. This class stub is the one that the Logic Module sends to the client for further use by the client application.

4.3 Prefetching Data at Runtime

When *dataClay* receives an execution request, data prefetching is started automatically due to analyzing the source code of the application, generating the prefetching hints and injecting prefetching instructions and methods done during the class registration

```java
public class BankManagement extends DataClayObject {

  public void setAllTransCustomers() {
    // Injected scheduling of prefetching lazy task
    dataService.addLazyTask(setAllTransCustomers_prefetch());

    for (Transaction trans : this.transactions) {
      trans.getAccount().setCustomer(this.manager);
    }
  }

  // Injected prefeteching method
  public void setAllTransCustomers_prefetch() {
    // Parallel prefetching of collection elements with Java 8
       Parallel Streams
    this.transactions.parallelStream().forEach(trans -> {
      trans.type.load();
      trans.emp.load();
      trans.account.cust.company.load();
    });

     // Cannot be parallelized
     manager.company.load();
  }

  // Injected method used to load final objects of access hints
  public void load() {

  }
```

Figure 4.6: *A complete example of the code of the class stub BankManagement.*

process. Figure 4.7 shows an overview of the different resources and communications that perform the data prefetching during the application execution.

When a client application uses *dataClay*, it executes methods from the stub classes sent by the Logic Module. The stubs in turn send execution requests to the Data Service where the object is stored in order to load the objects and execute the requested methods. When the execution request is received by the corresponding Data Service, the first step that is done is to schedule the prefetching method for execution in the queue of the Data Service's lazy task runner.

The lazy task runner checks the queue periodically to see if there are any methods pending execution. Once it finds the prefetching method in the queue, it starts executing it. At this point, the object owning the prefetching method is already loaded and

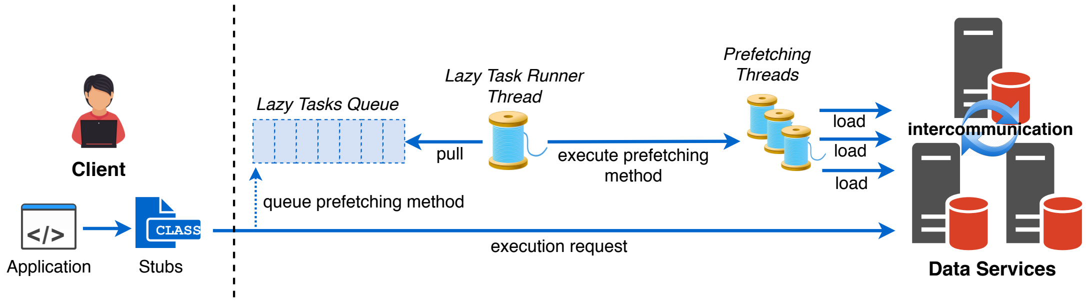


Figure 4.7: *A detailed view of the resources and communications that occur to perform data prefetching when an application is executed with dataClay.*

the prefetching method starts loading referenced objects as defined by its instructions. Given the structure of *dataClay*, when the prefetching method encounters an object in another Data Service, it communicates with that Data Service to load the object where it is stored. It is important to stress here that objects are never moved from one Data Service to another, even if the call to load an object comes from another Data Service.

For instance, if we have a *dataClay* deployment with three Data Services, DS_1, DS_2 and DS_3, as shown in Figure 4.1, and the client application wants to execute the method *setAllTransCustomers()*, shown in Figure 4.4, on an object of type *BankManagement* stored in Data Service DS_1, the steps followed by *dataClay* to perform the prefetching would be the following:

- First, the client application sends the execution request to *dataClay* through the stubs. The stubs automatically redirect the request to DS_1, where the object *BankManagement* is stored.

- When DS_1 receives the execution request of the method *setAllTransCustomers()*, it schedules the prefetching method *setAllTransCustomers_prefetch()* for execution by the lazy task runner.

- The lazy task runner periodically checks its queue for lazy tasks and starts executing *setAllTransCustomers_prefetch()* once it finds it in the queue.

- In this case, the first objects to load are the elements of the *transactions* collection. Since we are using parallel prefetching, the lazy task thread creates several subthreads and starts loading the elements of *transactions* in parallel, as explained in section 4.2.4. Given that the collection *transactions* is distributed among the three Data Services, each thread loads the elements and their related objects stored in a different Data Service. Thus, our approach further increases the improvement obtained from prefetching by loading the objects in parallel.

- When one of these threads, currently being executed on DS_1, tries to load an element stored in a different Data Service, say DS_2, *dataClay* redirects the load request to DS_2 and loads the object where it is stored.

4.4 Evaluation

We evaluated our approach by running experiments using the modified version of *dataClay* that includes our prefetching approach. All of the experiments were performed on the ITAN cluster at the Barcelona Supercomputing Center (BSC), which consists of 5 nodes that are interconnected by a 10GbE link. Each node is composed of a 4-core Intel Xeon E5-2609v2 processor (2.50GHz), with 32GB DRAM (DDR3), a 400GB SSD (Intel SSD P3500), and a 1TB HDD (WD10JPVX 5400rpm). We deployed *dataClay* on the cluster using one node as both the client and Logic Module, and 4 nodes to serve as 4 distinct Data Services.

To test the effectiveness of our prefetching approach, we used four well-known benchmarks from the object-oriented and Big Data fields: **(1)** OO7, **(2)** Wordcount, **(3)** K-Means, and **(4)** Princeton Graph Algorithms. We executed each benchmark 10 times and measured the average execution time. We also paid special attention to empty the cache after each execution, in order to guarantee that all executions are run on a cold cache without any objects.

We tested both the sequential as well as the automatically parallelized versions of our approach. We also compared the approach with the *Referenced-Objects Predictor (ROP)*, as defined in Section 2.3, using different fetch depths in all of the experiments. Before we explain our experimental results, we first go into an in-depth explanation of both the data models and computations of each benchmark.

4.4.1 Benchmarks

OO7

The data model of OO7 is an abstraction of different CAD/CAM/CASE applications and contains a recursive data structure involving complex inheritance and composition relationships, as depicted in Figure 4.8. The benchmark's has two main classes *Manual*, which contains text fields and is used to benchmark string operations, and *Module*, which is where all the data traversals of the benchmark start.

The benchmark contains a random data generator that takes as parameter the database size: small ($\tilde{1}$,000 objects), medium ($\tilde{3}$0,000 objects) and large ($\tilde{6}$00,000 objects). Moreover, OO7 has an implemented set of 6 "traversal operations":

- *t1:* tests the raw traversal speed by traversing the benchmark's complex data model, starting from the object *Module*.
- *t2a, t2b and t2c:* test the update speed by traversing the data model and updating different numbers of *Composite Parts* and *Atomic Parts* during the traversal.
- *t8 and t9:* test the text processing speed by scanning the *Manual* object and performing string operations on its contents.

We tested our prefetching approach in *dataClay* with the three different database sizes of OO7. As for the traversals, we executed *t1*, *t2a*, *t2b* and *t2c*. We did not run *t8*

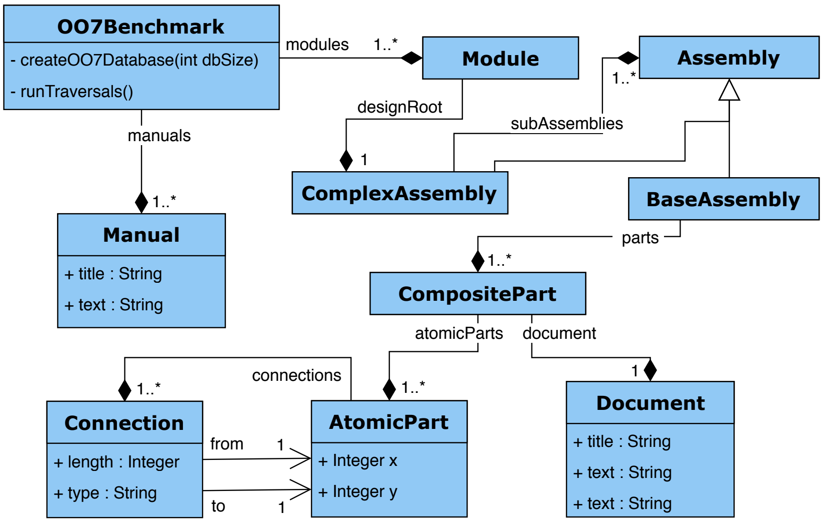


Figure 4.8: *Class diagram of the OO7 benchmark.*

or *t9* given that they are meant to test text processing speed and only load one object, *Manual*, with its string contents. Thus, these traversals are not designed to test data access speed, which is what can be improved using our prefetching approach.

Wordcount

Wordcount is a parallel algorithm that counts the appearances of all the different words within a set of files. The application parses the input files splitting their text lines into words and maintains a data structure to keep one counter per word. The application ends up producing a final output to present the counters for each unique word. Due to the resemblance of this algorithm to the problem of creating histograms, Wordcount is commonly used as a Big Data benchmark.

The data model of the Wordcount benchmark is depicted in Figure 4.9. Unlike OO7, the benchmark has a simple model consisting of several *Text Collections*, each containing one or more *Texts* representing the input files. Each of the *Text* objects in turn contains one or more *Chunks*, which represent fragments of the text that contain the final words to be counted.

We used Wordcount as a benchmark in order to test our prefetching approach on an application that does not contain many classes and relations between them. We ran the experiments using a data set of 8 files, containing a total of 10^7 words. We divided the texts into four collections and distributed the collections among the four Data Services of the *dataClay* deployment. Furthermore, in order to test our approach with different numbers of objects, we ran the benchmark with varying numbers of chunks per text, ranging from one chunk containing all the of the words in each text (i.e. few large objects) to 10^6 chunks containing very few words (i.e. many small objects).

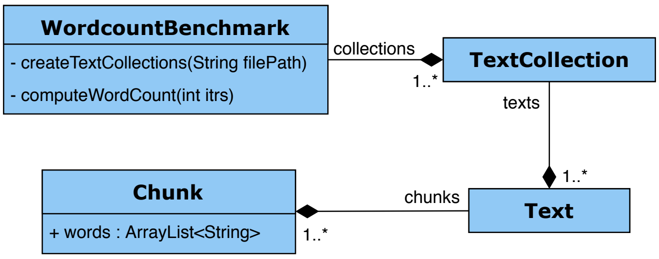

WordcountBenchmark
- createTextCollections(String filePath)
- computeWordCount(int itrs)
collections
1..*
TextCollection
texts
1..*
Chunk
+ words : ArrayList<String>
chunks
1..*
Text

Figure 4.9: *Class diagram of the Wordcount benchmark.*

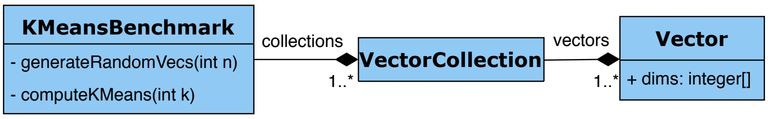


Figure 4.10: *Class diagram of the K-Means benchmark.*

K-Means

K-Means is a clustering algorithm commonly used as a Big Data benchmark that aims to partition n input vectors into k clusters in which each vector belongs to the cluster with the nearest mean. It is a complex recursive algorithm that requires several iterations to reach a converging solution. The data model of K-Means that we used, depicted in Figure 4.10, consists of a set of *VectorCollections* each containing a subset of the n input *Vectors*. Each vector contains a collection of integers, representing the vector's dimensions.

In our experiments, we ran the K-Means benchmark using various numbers of randomly generated vectors, n, each consisting of 10 dimensions, and different values of k. We did so to test the effect that the number of vectors, and hence persistent objects, has on the benefits obtained from our approach. We also divided the input vectors into 4 collections and distributed the collections among the *dataClay* Data Services.

Princeton Graph Algorithms

The Princeton Graph Algorithms (PGA) [74] is a collection of classes that is used to execute various complex graph traversal algorithms on different types of graphs (e.g. undirected, directed, weighted). Figure 4.11 depicts the subset of the benchmark's classes that we used in our experiments.

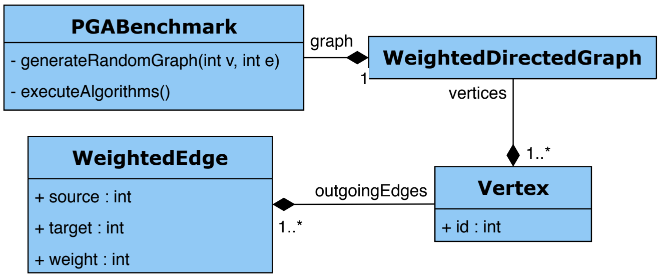


Figure 4.11: *Class diagram of the Princeton Graph Algorithms benchmark.*

We executed the Depth-First Search (DFS) and Bellman-Ford Shortest Path algorithms using a *WeightedDirectedGraph*. The graph consists of a set of *Vertex* objects, each containing the outgoing *WeightedEdges* of the vertex. We ran our experiments using different numbers of randomly generated vertices v and edges e, which we chose to construct graphs with different levels of edge density. As with the rest of the benchmarks, we distributed the data among the four Data Services of *dataClay*.

4.4.2 RQ5: Does our prefetching approach improve application execution times?

To answer this question, we measured the execution times of the studied benchmarks using *dataClay* without any prefetching, with our prefetching approach as well as with the ROP using varying fetch depths. The rest of this section details the experimental results for each of the benchmarks individually.

OO7

Figure 4.12 shows the execution times of the traversal *t1* of OO7. In this case, we used the ROP with fetch depths of between 1 and 10. The figure clearly offers more improvement to the original execution time of the benchmark than the ROP, even with a very high fetch depth of 10. Moreover, Figure 4.12 also indicates that parallelizing our approach adds additional reduction of OO7's execution time, offering an improvement ranging from 30% with the small database to 24% with the large one.

Figure 4.12 also shows that increasing the fetch depth of the ROP from 1 to 5 improves the application execution time, before it stagnates with only minor improvement observed when increasing the fetch depth to 10. This behavior is explained by the fact that ROP can only prefetch objects up to a certain depth before running out of referenced objects to prefetch. The exact depth at which ROP stagnates depends on the data model of the used application, as will become clearer when explaining the experimental results of the Wordcount benchmark in Section 4.4.2.

Another disadvantage of the ROP can also be seen in the results of our experiments. Figure 4.12 shows that prefetching data in parallel further increases the improvement

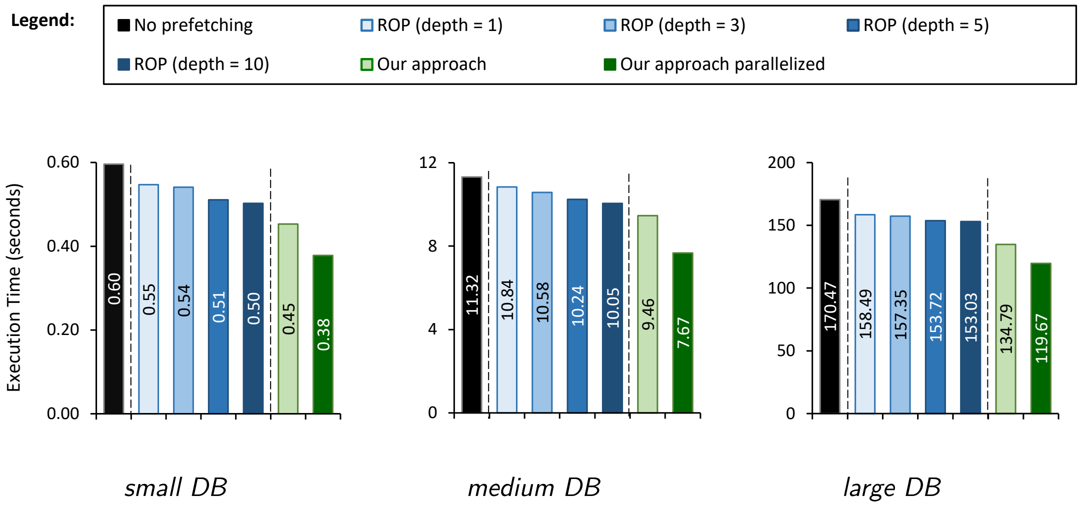


Figure 4.12: *Execution times of the traversal t1 of the OO7 benchmark without any prefetching, with ROP with different depths and with our prefetching approach in both sequential and parallel implementations.*

obtained from our approach. While we can automatically parallelize our approach, using the ROP to prefetch data in parallel is more difficult. This is due to the fact that the ROP only prefetches objects directly referenced from an accessed object and does not perform any collection prefetching, which is where our parallelized approach achieves the most gains in execution time.

When considering previous works on prefetching that have used OO7 as a benchmark, Ibrahim *et al.* report an improvement of 7% in execution time with the small OO7 database while Bernstein *et al.* report an improvement of 11% on the medium-sized database [6]. While these numbers are not directly comparable to the ones obtained in our experiments given that the approaches use a different POS, with different levels of optimization and run their experiments on different hardware, it is worth mentioning that our approach achieves an improvement of 30% and 26% with the small and medium OO7 databases respectively.

We also found an interesting behavior of our approach with the traversals *t2a*, *t2b* and *t2c* of OO7. Unlike the traversal *t1*, these traversals do not traverse the entire object graph of the benchmark but rather stop when reaching certain objects, and then proceed to perform an update operation on these objects. In this thesis, we report the execution times of the traversal *t2b* but both *t2a* and *t2c* exhibit similar behaviors.

Figure 4.13 shows the execution time of the traversal *t2b*, which performs an update

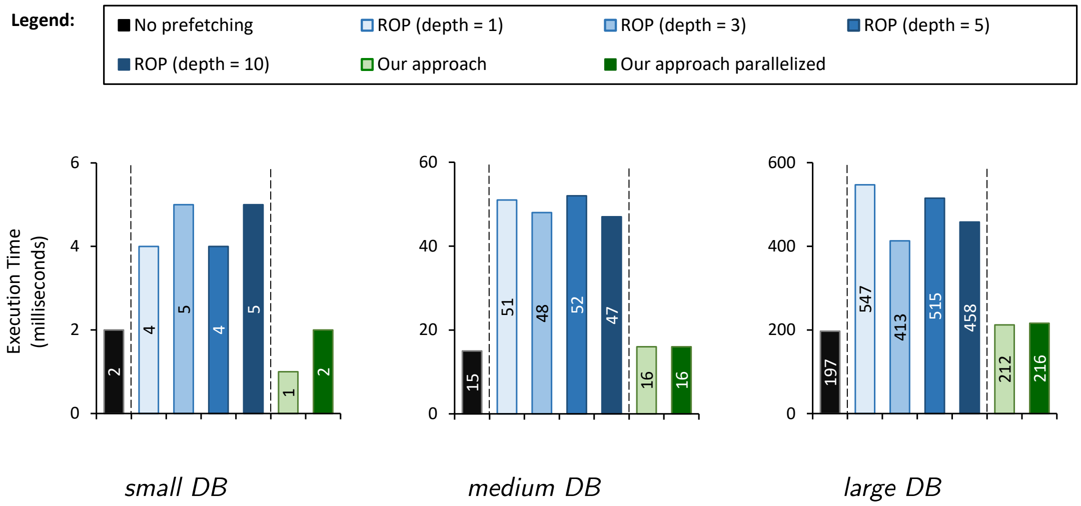
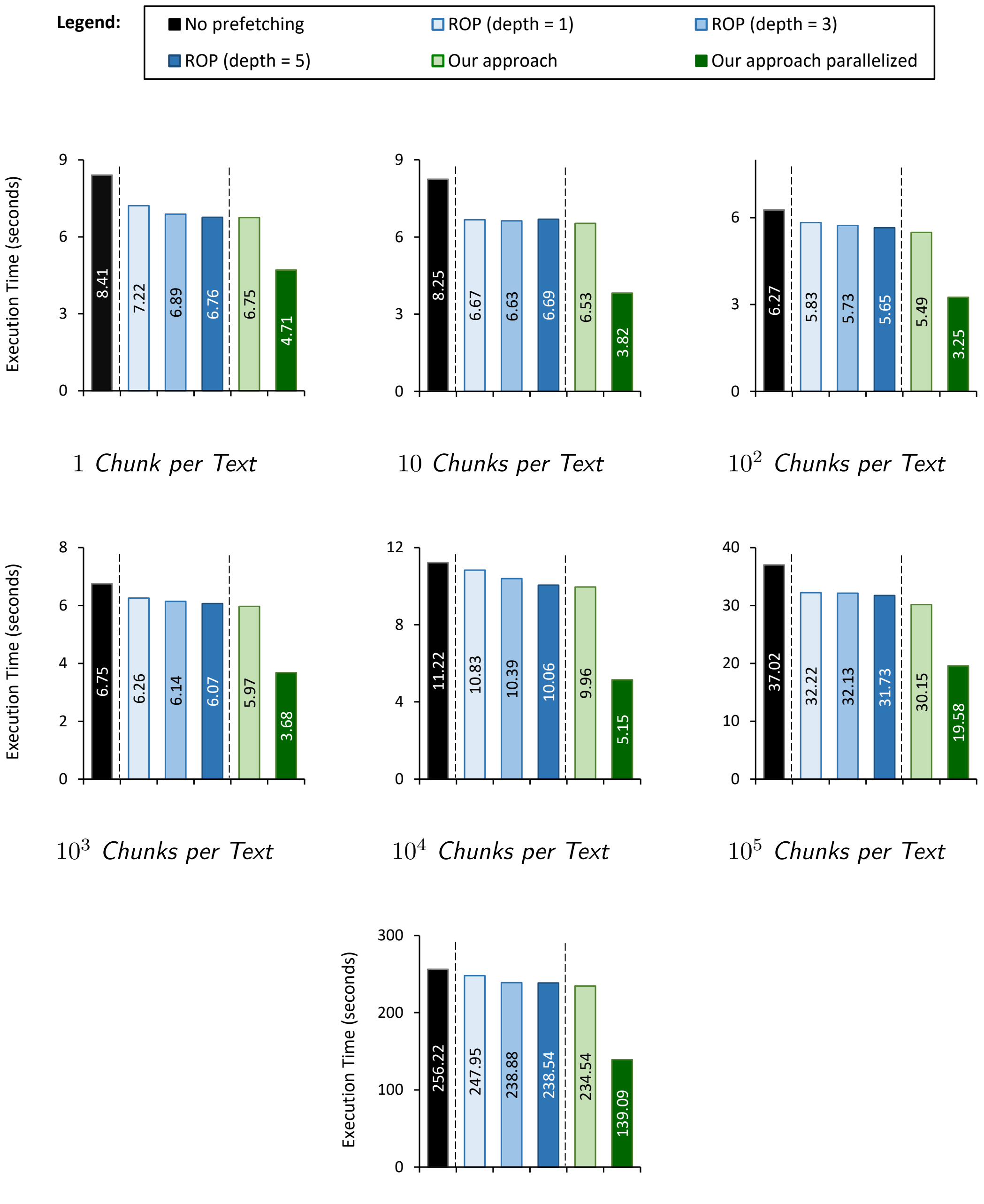


Figure 4.13: *Execution times of the traversal t2b of the OO7 benchmark without any prefetching, with ROP with different depths and with our prefetching approach in both sequential and parallel implementations.*

operation on 10% of the benchmark's objects. The execution times in the figure are shown in milliseconds, because this traversal is executed much faster than *t1*. This is due to the fact that that some of the objects it accesses remain in memory from the earlier execution of *t1*. We can see from Figure 4.13 that our prefetching approach does not offer any improvement in the execution time of this traversal, which is due to the fact that the latency of the traversal is not caused by data access times but rather by the time taken to update the objects.

In the case of the small OO7 database, it appears that the sequential version of our approach performs better than the parallelized version. However, the difference is only 1 millisecond and can be attributed to any impurities at the time of calculation. In reality, using the sequential or parallelized versions of our approach should result in the same execution time of *t2b*, as with the medium and large databases in Figure 4.13.

On the other hand, Figure 4.13 also shows that using the ROP adds extreme overhead, equivalent to doubling the execution time of *t2b* in some cases. This overhead is caused by the fact that the ROP in this cases prefetches the objects referenced from the object being updated, oblivious of the fact that these objects are in fact never accessed later on in the application. This is in contrast to our approach that takes into consideration the application's code and is aware that these objects are never accessed, and hence it never predicts them to be prefetched.



1 *Chunk per Text* 10 *Chunks per Text* 10^2 *Chunks per Text*

10^3 *Chunks per Text* 10^4 *Chunks per Text* 10^5 *Chunks per Text*

10^6 *Chunks per Text*

Figure 4.14: *Execution times of the Wordcount benchmark without any prefetching, with ROP with different depths and with our prefetching approach in both sequential and parallel implementations.*

Wordcount

Figure 4.14 shows the execution times of the Wordcount benchmark. We can see that the ROP in this case stagnates with a fetch depth of 5, not offering any improvement with larger depths. We can also see that the benchmark's execution time using our prefetching approach in its sequential implementation is almost identical to that obtained with the ROP with a fetch depth of 5. This is due to the fact that, unlike OO7, Wordcount has a fairly simple data model where applying the ROP with an adequate depth can indeed prefetch all the necessary objects.

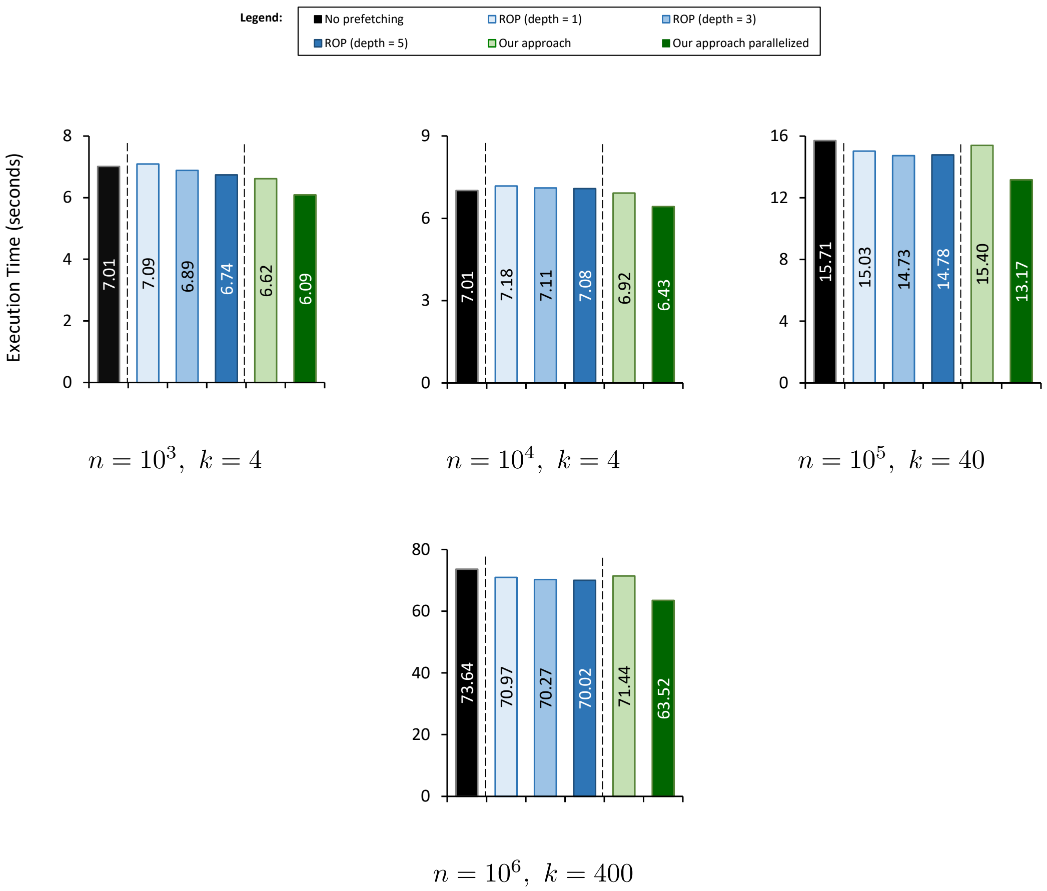
However, the big advantage of using our approach comes with the parallel prefetching that it offers. Figure 4.14 indicates that our parallelized approach offers big improvement in the application execution time, cutting it by more than 50% in some cases. This improvement is considerably higher than what we obtained with OO7, again due to the differences between the data model of the two benchmarks. In the case of Wordcount, the data model contains many collection associations, which can be easily prefetched in parallel, and few single associations between objects, which are more common in OO7 and cannot be prefetched in parallel.

Finally, Figure 4.14 also shows that increasing the number of chunks per text up to 10^6 does not affect the improvement obtained by our approach. This indicates that our approach offers stable improvement in application execution times, whether used with applications that handle a small number of large objects or many small-sized objects.

K-Means

Figure 4.15 shows the execution times of the K-Means benchmark. In this case, the ROP does not offer any significant improvement regardless of the fetch depth given that the benchmark's data model does not contain any single associations that can be prefetched. Similarly, our approach in its sequential implementation does not offer any improvement given that the benchmark's algorithm retrieves all of the manipulated data (i.e. the vector collections and vectors) at the beginning of the execution.

However, our approach achieves better improvement, reducing between 9% and 15% of the benchmark's execution time, when prefetching data in parallel. This is due to the same reason as with the Wordcount benchmark, since the model of the K-Means contains a collection association that is easily prefetched in parallel, which is what causes the higher improvement in this case. Finally, Figure 4.15 indicates that



$n = 10^3,\ k = 4$

$n = 10^4,\ k = 4$

$n = 10^5,\ k = 40$

$n = 10^6,\ k = 400$

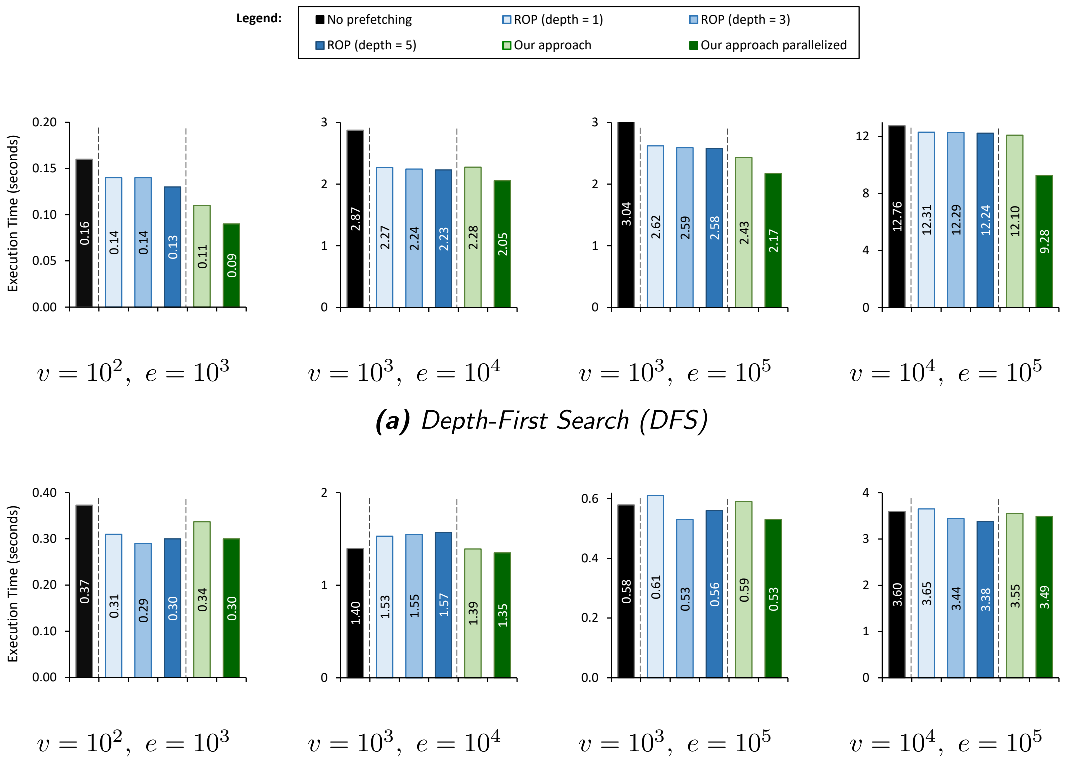
Figure 4.15: *Execution times of the K-Means benchmark without any prefetching, with ROP with different depths and with our prefetching approach in both sequential and parallel implementations.*

the improvement that our approach offers is not affected by the size of the dataset manipulated by the benchmark.

Princeton Graph Algorithms

Figure 4.16 (a) shows the execution times of the Princeton Graph Algorithms benchmark. In particular, Figure 4.16 (a) shows the execution times of the DFS algorithm and indicates that neither our approach in its sequential implementation nor the ROP offer significant improvement. This is because of the same reasons as the K-Means benchmark, the DFS algorithm loads the entire data set at the beginning of execution and does not allow enough time for the prefetching to occur. However, when considering the parallel version, our approach results in an improvement of around 30%.

On the other hand, Figure 4.16 (b) indicates that even when using our approach in



$v = 10^2,\ e = 10^3$ $v = 10^3,\ e = 10^4$ $v = 10^3,\ e = 10^5$ $v = 10^4,\ e = 10^5$

***(a)** Depth-First Search (DFS)*

$v = 10^2,\ e = 10^3$ $v = 10^3,\ e = 10^4$ $v = 10^3,\ e = 10^5$ $v = 10^4,\ e = 10^5$

***(b)** Bellman-Ford Shortest Path*

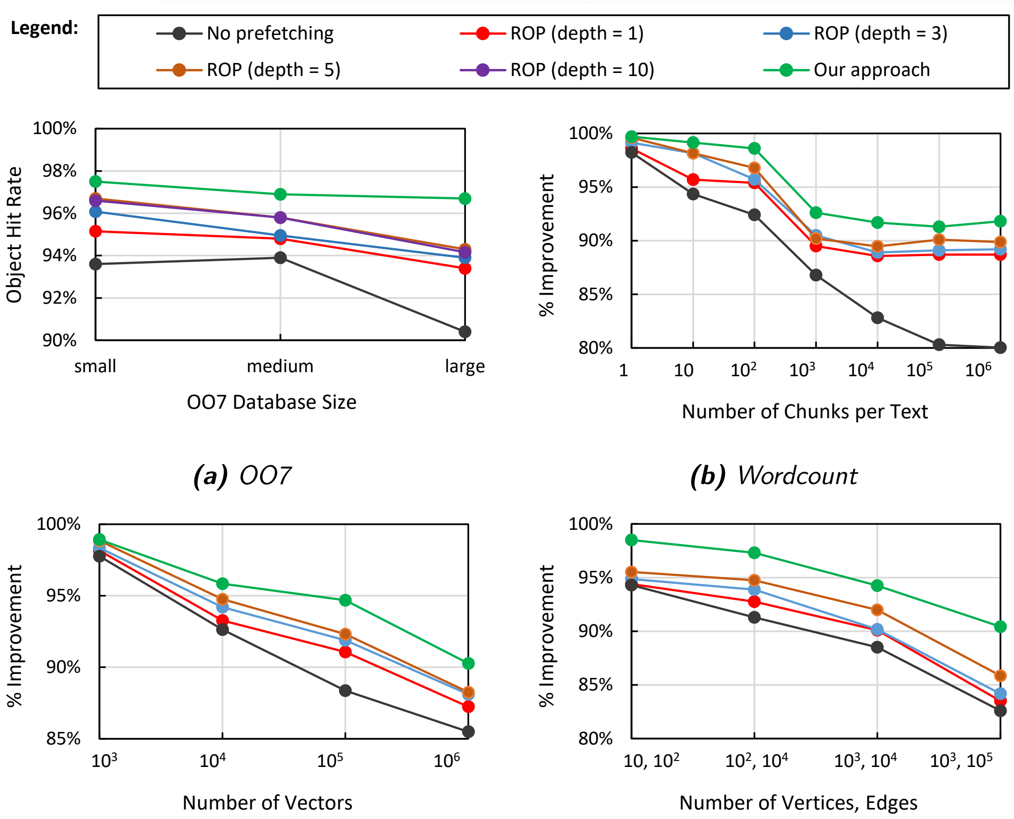
***Figure 4.16:** Execution times of the Princeton Graph Algorithms benchmark without any prefetching, with ROP with different depths and with our prefetching approach in both sequential and parallel implementations.*

parallel, we do not see significant improvement in the execution time of the Bellman-Ford algorithm. This is due to the fact that this algorithm does not access the graph's vertices in a predetermined order, but rather starts from a source vertex and applies a trial-and-error approach to reach the shortest path solution using various intermediate data structures, and thus predicting access to the objects it uses is more difficult. Evidently, the ROP does not offer any improvement in this case either.

4.4.3 RQ6: What is the object hit rate of the prefetching approach?

We answered this question by calculating the percentage of objects that are already found in memory when they are accessed by the studied benchmarks. Figure 4.17 shows this object hit rate of our approach compared with the original hit rate without any prefetching and the hit rate of ROP with various fetch depths.

Figure 4.17 indicates that using our approach, we can obtain a higher object hit



(a) *OO7* ***(b)*** *Wordcount*

(c) *K-Means* ***(d)*** *Princeton Graph Algorithms*

Figure 4.17: *The object hit rate obtained in the benchmarks without any prefetching, with ROP with different depths and with our prefetching approach.*

rate than the ROP. In the case of OO7, the object hit rate obtained with our approach ranges between 97% and 98% whereas with Wordcount it decreases from 99% when using 1 chunk per text to 92% with 10^6 chunks per text. This decrease is expected due to the large number of objects when increasing the number of chunks per text. A similar behavior can be observed in the K-Means and Princeton Graph Algorithms as well, where the object hit rate achieved by our approach decreases when increasing the number of objects manipulated by the benchmarks.

In the case of K-Means and PGA, we also notice a slightly lower improvement in the hit rate of our approach. This is due to the data model and the nature of the benchmarks, which also cause a lower improvement in the execution time of both benchmarks, as explained in Section 4.4.2. However, in all of the benchmarks, we consistently achieve a higher hit rate than the ROP, regardless of the used fetch depth.

Looked at from a different point of view, Figure 4.17 indicates that, for instance, in the case of the small OO7 database, our prefetching approach increases the object hit rate from 93.6% without any prefetching to 97.5%. This in turn means that our approach reduces the object miss rate from 6.4% to 2.5%, a decrease of 60.94%. This decrease in the rate of objects not found in memory when accessed by the benchmarks is what results in the improvement of the benchmark execution time when using prefetching, as explained in Section 4.4.2.

4.5 Summary

In this chapter, we demonstrated the benefits of prefetching based on static code analysis of object-oriented applications by integrating our prefetching approach into *dataClay*, a POS developed for the purpose of data distribution and sharing. We integrated our approach into *dataClay* by implementing various changes to its source code to allow automatic prefetching of data with the execution requests received by the system. We also further optimized the prefetching approach by parallelizing the prefetching methods, allowing data to be prefetched from various *dataClay* Data Services in parallel. This parallelization process is fully automatic and does not change the application's original functionality nor requires any input from the clients.

We tested the improvement that can be obtained by using our prefetching approach on a set of four well-known benchmarks. The experimental results indicate that our approach offers the highest improvement in execution time when used with applications with a complex data model, such as OO7. Moreover, prefetching data in parallel proved extremely beneficial, especially with simple data models that contain many collection associations, such as the case with the Wordcount and K-Means benchmarks. We also encountered one limitation of our approach, with the Bellman-Ford shortest path algorithm, where it could not offer significant improvement because the algorithm accesses persistent objects in a random order that is difficult to predict.

To handle this type of cases, we can augment our approach with more information obtained during application execution to make better predictions on which data should be prefetched (e.g. the most connected nodes in a graph are more likely to be visited and hence should be prefetched). However, such a hybrid approach might introduce overhead to application execution times and thus should be studied in more details to determine if the gains we obtain compensate the introduced overhead.

In terms of data size, our approach provides the same level of improvement regardless of the number or size of persistent objects manipulated by each benchmark. When compared with the ROP, our approach achieves at least the same improvement and, in cases where prefetching is not needed, has less of a negative effect on application performance. When considering the parallel version, our proposed always reduces the execution times of the benchmarks by a higher percentage than the ROP.

The results discussed in this chapter indicate that using static code analysis of object-oriented applications to predict data for prefetching in POSs does not only offer theoretically high-accuracy prediction, but also improves the execution times of applications. We thus demonstrate the viability of our first proposed approach to analyze statements and instructions of computer languages to predict data for prefetching.

Chapter 5

Prefetching in RDF Triplestores

This chapter discusses the last contribution of this thesis **(C3)** and presents an approach to analyze queries of the declarative query language SPARQL in order to predict data for prefetching from RDF Triplestores. Ideally, we planned to perform a static analysis approach similar to the one we used to analyze OO applications to prefetch data from POSs. However, unlike the case with POSs, where the entire application code is known beforehand, queries are received consecutively by the SPARQL endpoints of RDF Triplestores and are not known in advance.

Thus, we could not perform the predictions statically and needed to adopt another approach that can predict the upcoming queries that will be executed. We achieved this by analyzing historic logs of SPARQL queries to detect recurring query patterns and using the detected patterns to predict upcoming queries. We then prefetch the data that the queries will access from the RDF Triplestore by applying query augmentation, a technique that aims at rewriting queries by relaxing their conditions in order to retrieve data that is potentially relevant to several subsequent requests at once.

We start the chapter by defining some RDF and SPARQL preliminaries in Section 5.1. We then introduce a motivating example that shows the advantages of our approach in Section 5.2. Afterwards, we formalize our approach in Section 5.3 and explain how we use machine learning algorithms to predict the structure and content of the augmented query used to prefetch data. Thereafter, we evaluate our approach and demonstrate its viability in Section 5.4 by answering the following research questions:

RQ7: What is the prediction accuracy of the proposed query-log analysis approach??

RQ8: Can the predictions be made within a reasonable amount of time?

RQ9: What is the cache hit rate of the proposed prefetching approach?

Finally, we summarize the chapter and conclude our findings in Section 5.5.

5.1 RDF and SPARQL Preliminaries

The **Resource Description Framework (RDF)** is a simple, extensible graph data model for representing semantically-rich information on the web [81]. Its main structure is *triples* that link two resources, the *subject* and *object*, through a property, the *predicate* of the triple. For instance, an example of such a triple is `dbr:Iker_Casillas dbo:team dbr:Real_Madrid` which indicates that the resource *Iker Casillas* is linked to the resource *Real Madrid* through the property *team*.

The set of all RDF triples in a RDF Triplestore form the store's data graph. The resources constitute the nodes of the graph while the properties constitute its edges. The data graphs stored in RDF Triplestores can be accessed by a variety of means, the most common of which is the SPARQL query language.

SPARQL is a high-level, declarative, semantic query language. SPARQL queries have four different forms, namely SELECT, DESCRIBE, ASK and CONSTRUCT. Previous studies show that the most common query starts with one or more PREFIX items followed by a SELECT structure [59, 62]. Therefore, in our approach we only consider SPARQL queries of the SELECT form and we do not study the less common forms.

The central construct of a SPARQL SELECT query is a triple pattern. A **Triple Pattern** is defined as $T = \langle s, p, o \rangle \in (V \cup U) \times (V \cup U) \times (V \cup U \cup L)$ where V is a set of variables, U a set of URLs and L a set of literals [71]. The three parts of a triple pattern in a SELECT query correspond to a subject, a predicate and an object.

A set of one or more triple patterns constitute a **Basic Graph Pattern (BGP)**. A SELECT query can contain one or more BGPs, joined with the SPARQL keywords AND, UNION or OPTIONAL. These BGPs form the query's graph pattern. When a SPARQL query is evaluated against an RDF triplestore, the query's graph is compared to the data graph of the store and the data subgraph that matches the query's graph is returned as the query results. For instance, Figure 5.1, to be explained in Section 5.2, shows a sequence of SPARQL SELECT queries.

In order to launch SPARQL queries to an RDF Triplestore, clients use a **SPARQL Endpoint**, which is a service that executes SPARQL queries against a defined triplestore and returns the corresponding results [75]. We call a consecutive sequence of queries received by a SPARQL endpoint from the same client a **Query Session**. As previous studies have demonstrated, queries in the same query session tend to be similar to each other with only minor changes occurring between them [26, 72]. While there is no standard definition for the length of a query session, we follow the most common approach of defining the sessions to be one-hour long [54, 55, 84].

5.2 Motivating Example

Figure 5.1 shows two query sessions received by a SPARQL endpoint that consist of three and two SELECT queries, respectively. The queries in the first session look up former teams of different football players and ask for some properties of these teams whereas the queries in the second session look up actors and the movies they have starred in. We use the line numbers in the figure to refer to the triple patterns of the queries. For instance, on line 4 we can see the triple pattern `dbr:Iker_Casillas dbo:formerTeam ?team`, so we will refer to this triple pattern as T_4.

In this example, the query graph of Q_1 consists of a single BGP with one triple pattern, whereas the query Q_3 also has a single BGP but consisting of two triple patterns. On the other hand, the query graph pattern of Q_2 is more complex, consisting of two BGPs, each with one triple pattern, connected using the keyword OPTIONAL.

We can see that the triple patterns of the queries of the first session in Figure 5.1 (a) are quite similar to each other. For instance, T_{20} is identical to T_{12} whereas T_{10} only differs from T_4 in the subject, but has the same predicate and object. Similarly, the queries in the second session in Figure 5.1 (b) are also similar to each other with only minor changes occurring between their triple patterns.

Previous approaches to prefetching, such as [54] and [84], take advantage of these similarities in order to prefetch data for subsequent queries. However, there is another

```
Q1 : PREFIX dbr: <http://dbpedia.org/resource/>
     PREFIX dbo: <http://dbpedia.org/ontology/>
     SELECT * WHERE {
       dbr:Iker_Casillas  dbo:formerTeam  ?team .
     }

Q2 : PREFIX dbr: <http://dbpedia.org/resource/>
     PREFIX dbo: <http://dbpedia.org/ontology/>
     SELECT * WHERE {
       dbr:Cristiano_Ronaldo  dbo:formerTeam  ?team .
       OPTIONAL {
         ?team  dbo:manager ?manager .
       }
     }

Q3 : PREFIX dbr: <http://dbpedia.org/resource/>
     PREFIX dbo: <http://dbpedia.org/ontology/>
     SELECT * WHERE {
       dbr:Gerard_Pique  dbo:formerTeam  ?team .
       ?team  dbo:manager  ?manager .
     }
```

***(a)** Session 1*

```
Q4 : PREFIX dbr: <http://dbpedia.org/resource/>
     PREFIX dbo: <http://dbpedia.org/ontology/>
     SELECT * WHERE {
       dbr:Richard_Gere  dbo:starring  ?movie .
     }

Q5 : PREFIX dbr: <http://dbpedia.org/resource/>
     PREFIX dbo: <http://dbpedia.org/ontology/>
     SELECT * WHERE {
       dbr:Jack_Nicholson  dbo:starring  ?movie .
       OPTIONAL {
         ?movie  dbo:director ?director .
       }
     }
```

***(b)** Session 2*

***Figure 5.1:** Example query session of SPARQL SELECT queries*

type of similarity that can be useful in this case. Looking across the two sessions, we can see that the structures of the first two queries are the same: both sessions start with a query with a single triple pattern then a second query with two triple patterns and the keyword OPTIONAL. Moreover, although the two sessions treat semantically different topics, the same changes occur between the triple patterns of the consecutive queries. For instance, both T_{10} from Session 1 and T_{31} from Session 2 result from changing the subjects of the triple patterns T_4 and T_{25}, respectively, and maintaining the same predicate and object.

Hence, by analyzing previous query sessions of the SPARQL endpoint we can detect recurring patterns of query structures as well as triple patterns that can be then used to prefetch data for subsequent queries in the current session. For instance, if we analyze the queries in Session 1, we can detect the patterns present between the structures of the consecutive queries as well as the changes that led to their triple patterns. We can then use these detected patterns to *predict* the structure of the next query in Session 2 as well as what triple patterns already present in the session will be used in it. Using this prediction, we can then launch the query before it is received, hence prefetching its results before the client asks for them.

Due to the large number of resources in repositories, it is very difficult to predict exactly which resource will be used in a query. For example predicting that the resource *dbo:Jack_ Nicholson* in Q_5 will be used after receiving the query Q_4 with the resource *dbo:Richard_ Gere*. However, we can predict which part of the triple pattern is changed among the queries. Thus, instead of predicting the exact next query in a session, we relax the conditions resulting from the query's triple patterns and construct an augmented query that prefetches data relevant to subsequent queries in the session.

5.3 Proposed Approach

The main goal of our approach is to predict data relevant to subsequent queries received by a SPARQL endpoint. We do so by training a machine learning model on historic query logs to detect recurring patterns in previous query sessions. However, instead of using the queries received by the endpoint directly, we use two more abstract structures that capture the similarities between SPARQL queries at a higher level: Query Types that capture the structural similarity between queries and Triple-Pattern Mappings that capture the similarities between the triple patterns of the queries. Based on these similarities, we construct augmented queries to prefetch relevant data.

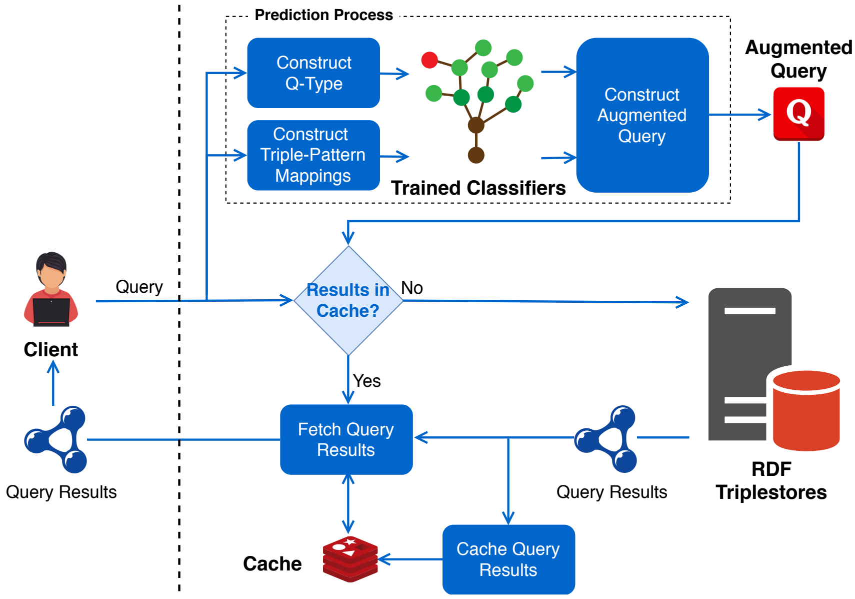


Figure 5.2: *Design of a prefetching and caching system for SPARQL endpoints using our approach.*

Figure 5.2 shows a proposed design of a prefetching and caching system using our approach. When a new query arrives to the SPARQL endpoint, the system first checks the cache to see if the query results have already been prefetched by a previously predicted augmented query. If this is the case, the query results are retrieved from the cache and returned to the client directly without executing the query.

The system then computes the query's Query Type and Triple-Pattern Mappings and runs these structures through the trained machine learning models. It then uses the prediction results to construct a new augmented query. Before executing the augmented query, the system checks if the query is a duplicate of a previous augmented query. If it is the case, then the query results are already cached and there's no need to execute the query again. Otherwise, the augmented query is executed to prefetch data relevant to subsequent queries in the session and its results are stored in the cache.

In the rest of this section, we first describe how we construct Query Types that capture the structural similarity between SPARQL queries in Section 5.3.1. Second, we detail how we construct Triple-Pattern Mappings to capture the content similarity between the triple patterns of SPARQL queries in Section 5.3.2. Afterwards, we describe how we train a machine learning model using these structures in order to capture the repetitive patterns in previous query sessions received by the endpoint in Section 5.3.3.

We then move on to discuss how the trained models are used to predict which Query Type and Triple-Pattern Mappings should be used to construct the augmented query in Section 5.3.4. Finally, we describe how the results of the augmented queries can be cached for use with subsequent queries in the session in Section 5.3.5. It should be noted that we do not implement the physical cache in this thesis, but rather we focus on the prediction and construction process of the augmented queries. We do however provide an estimation of the 'cache hit rate' that can be obtained by utilizing our approach in Section 5.4.3.

5.3.1 Query Types

The aim of a 'query type', also denoted **Q-Type**, is to capture the syntactic structure of SELECT queries. We compute the Q-Type of a given query by generating the query's parse tree (following the SPARQL 1.1 grammar), removing the leaves of the tree and serializing the resulting tree. We denote '**surface form**' to the leaves of the tree, and '**inner tree**' to the rest of the tree. Therefore, we say that two queries have the same Q-Type, and hence are structurally similar, if they differ only in their *surface form*. That is, they have the same *inner tree* but different variable names, resources and literals in their *surface form*.

Example. Performing the parsing process described above on the SPARQL query Q_1 in Figure 5.1 (a) results in the parse tree shown in Figure 5.3. This parse tree consists of two types of nodes: leaf nodes and inner nodes. On the one hand, leaf nodes (in orange) represent the *surface form* of the query, that is the actual text appearing in the query. On the other hand, inner nodes (in blue) represent the *inner tree* of the query, that is the Q-Type that abstracts the structure of the query from its content and captures its structure. For instance, the query Q_4 from Figure 5.1 (b) would have the same Q-Type as Q_1, since both queries have the same *inner tree* and only differ in the variables, resources and properties that appear in their *surface form*.

On the other hand, a more complex example is shown in Figure 5.4, which depicts the parse tree of the query Q_2 from Figure 5.1 (a). This query consists of two *BGPs*, each of one triple pattern, connected together with the keyword OPTIONAL. As we can

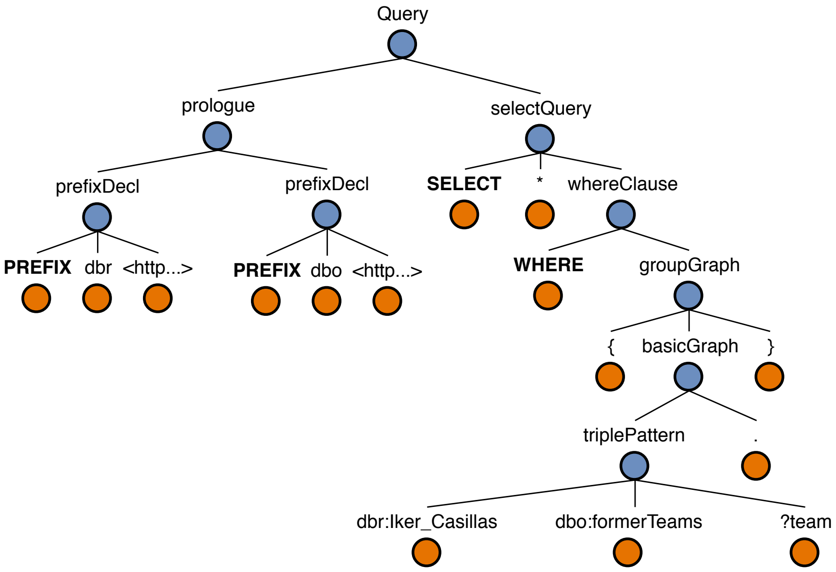


Figure 5.3: *Parse tree of the query Q_1 from Figure 5.1. Nodes in orange represent the surface form of the query while nodes in blue represents its inner tree.*

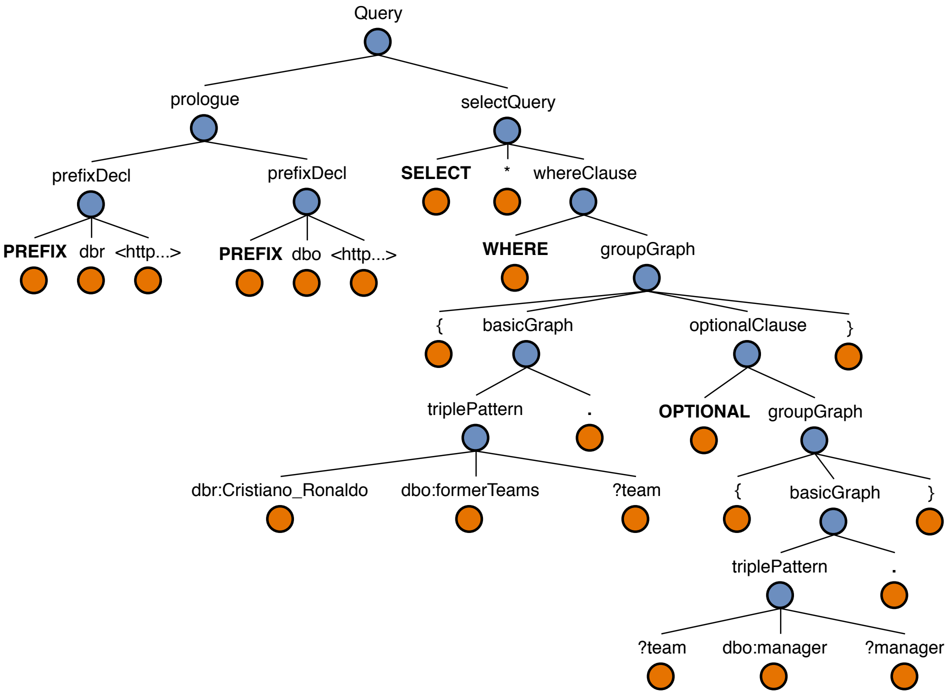


Figure 5.4: *Parse tree of the* SELECT *query Q_2 from Figure 5.1. Nodes in orange represent the surface form of the query while nodes in blue represents its inner tree.*

see, the Q-Types capture the structure of a SPARQL query, including how its triple patterns form BGPs and, when necessary, how the BGPs connect with each other using the keywords AND, UNION and OPTIONAL.

5.3.2 Triple-Pattern Mappings

In order to capture the changes that occur between the triple patterns of consecutive queries in a query session, we map each triple pattern to the most similar triple pattern that already appeared in a previous query. We do so by counting the number of triple pattern parts (i.e. subjects, predicates and objects) that are different between two triple patterns. In this measure, we say that two triple pattern parts are identical, and hence their distance is 0, if they are both variables or have the same URL or literal. Otherwise, we say that their distance is 1. More formally, assuming that x_1, x_2 are either the subjects, predicates or objects of two triple patterns $T_1 = \langle s_1, p_1, o_1 \rangle$ and $T_2 = (s_1, p_1, o_1)$, we define the distance between the two parts $\Delta(x_1, x_2)$ as:

$$\Delta(x_1, x_2) = \begin{cases} 0, & if\ (x_1 \in V \land x_2 \in V) \lor (x_1 = x_2) \\ 1, & otherwise \end{cases} \tag{5.1}$$

We then determine the overall distance between the two triple patterns $\Delta(T_1, T_2)$ by aggregating the individual triple pattern part distances as follows:

$$\Delta(T_1, T_2) = \Delta(s_1, s_2) + \Delta(p_1, p_2) + \Delta(o_1, o_2) \tag{5.2}$$

This function is based on the distance function defined by Lorey *et al.* [54]. In the original definition, the authors use a Levenshtein distance to compare two URLs or literals when measuring the distance between two triple pattern parts $\Delta(x_1, x_2)$ and then use a more complex aggregation to compute $\Delta(T_1, T_2)$. We modified it in our approach since we are only interested in counting the number of different triple pattern parts between T_1 and T_2, regardless of whether they are variables, URLs or literals.

By contrast, we introduce a restriction not found in the original definition to guarantee that the matched triple patterns are not too different from each other. We do so by limiting the distance between the mapped triple patterns to $\Delta(T_1, T_2) \leq 1$, i.e. the two triple patterns are different in at most one part. If no such match can be found, we say that the triple pattern is “unmapped”.

Afterwards, we create a **Triple-Pattern Mapping** from T_2 to T_1 by indicating that T_2 can be constructed from T_1 with a change in the specified triple pattern part x (or ϕ if the mapped triple patterns are identical). This can be formalized as follows:

$$T_2 = (T_1, x) \; where \; x \in \{s, p, o, \phi\} \tag{5.3}$$

For completeness, we define the mapping of an unmapped triple pattern $T_{unmapped}$ to be the same triple pattern without any part change $T_{unmapped} = \langle T_{unmapped}, \phi \rangle$.

Example. Looking at the queries Q_1 and Q_2 from Figure 5.1 (a), we say that the triple pattern T_{10} = `dbr:Cristiano_Ronaldo dbo:formerTeam ?team` is mapped to T_4 = `dbr:Iker_Casillas dbo:formerTeam ?team` with a distance of 1; they only differ in the subject but have the same predicate and object and thus $T_{10} = (T_4, s)$. On the other hand, T_{12} = `?team dbo:manager ?manager` is unmapped to either triple pattern, since it has a distance greater than one to both of them.

Similarly, the triple pattern T_{19} = `dbr:Gerard_Pique dbo:formerTeam ?team` of the query Q_3 is mapped to both T_4 and T_{10} with the same distance and can be rewritten as: $T_{10} = (T_4, s)$. Finally, The triple pattern T_{20} is identical to T_{12} and hence they are mapped together with a distance of 0 and is rewritten as: $T_{20} = (T_{12}, \phi)$

5.3.3 Machine Learning Models

After constructing the Q-Types and Triple-Pattern Mappings, as described in Section 5.3.1 and Section 5.3.2 respectively, we construct our machine learning models using these two structures in order to predict which Q-Types and Triple-Pattern Mappings should be used in the augmented query. The prediction can be seen as a time series forecasting problem, where the input is a sequence of ordered data points used to predict the value of a future instance [12]. In this case, the data points represent the Q-Types and Triple-Pattern Mappins of previous queries and the future instance to predict is the next query. Time series analysis is often reduced to a multi-class classification problem, which is the problem of classifying instances into one of three or more classes [3].

As such, we formulate the prediction as a multi-class classification problem. In this case, the classes are the different Q-Types and Triple-Pattern Mappings that are found in a query session. We used one classifier for the Q-Type and one classifier for *each* Triple-Pattern Mapping in the query. For the Q-Type classifier, the feature vector includes the Q-Types of the previous queries in the session. As for the triple pattern classifiers, we use as features the Triple-Pattern Mappings of the previous queries in the session, regardless of the position of the triple pattern in the original query.

We then train these models on the historic query logs of the SPARQL endpoint to capture the repetitive patterns of Q-Types and triple patterns. These models can also be retrained periodically, e.g. once a day or a week, to take into account new patterns that might have occurred. Moreover, since the training of the models only depends on query logs, it can be done offline in order to avoid affecting the endpoint's response time to clients while the models are being updated.

Example. Using the mappings explained in Section 5.3.2, we can rewrite the queries from Figure 5.1 (a) as follows:

$$\begin{aligned}
&Q_1 = (qtype_1, \{(T_1, \phi)\}) \\
&Q_2 = (qtype_2, \{(T_1, s), (T_2, \phi)\}) \\
&Q_3 = (qtype_3, \{(T_1, s), (T_2, \phi)\}) \\
&where:\ T_1 = \texttt{dbr:Iker\_Casillas dbo:formerTeam ?team} \\
&\qquad\quad\ T_2 = \texttt{?team dbo:manager ?manager}
\end{aligned}$$

As we can see, by rewriting the queries in a session using their Q-Types and Triple-Pattern Mappings we reduce the number of unique triple patterns, renumbered T_1 and T_2 here. We use this form of the queries to train the classifiers described earlier in this section to capture the changes that occur between consecutive queries in the session.

5.3.4 Constructing Augmented Queries

After training the classifiers, as described in Section 5.3.3, when a new query arrives to the endpoint, we calculate its Q-Type and Triple-Pattern Mappings and run this information through the trained classifiers. We then use the predicted Q-Type and Triple-Pattern Mappings to construct the augmented query used to prefetch data.

Example. Performing the same process to generate the Q-Types and Triple-Pattern Mappings on the queries Q_4 and Q_5 in Figure 5.1 (b) results in the following:

$$Q_4 = (qtype_1, \{(T_1, \phi)\})$$

$$Q_5 = (qtype_2, \{(T_1, s), (T_2, \phi)\})$$

$$where: \; T_1 = \texttt{dbr:Richard\_Gere dbo:starring ?movie}$$

$$T_2 = \texttt{?movie dbo:director ?director}$$

Hence, if we train our classifiers on the queries of Session 1, and run the queries in Session 2 through the trained models, the next query in the session predicted by the classifiers would be $Q'_6 = (qtype_3, \{(T_1, s), (T_2, \phi)\})$. This prediction indicates that the next query in Session 2 will be of Query-Type $qtype_3$ and have two triple patterns: **(1)** the first resulting from the triple pattern T_1 with a change in subject, and **(2)** the second identical to the triple pattern T_2.

Given that the first triple pattern results from a change in the subject of T_1, we replace the subject $dbo : Richard\_Gere$ with a variable $?var_1$ to construct the augmented query. This is done so that the new triple pattern allows the query to prefetch additional data relevant to several subsequent queries of the session. On the other hand, the models predict that the second triple pattern of the query is identical to T_2 and no substitutions are needed here. Combining this prediction with the Q-Type prediction, we obtain the predicted augmented query shown in Figure 5.5.

```
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT * WHERE {
   ?var1  dbo:starring  ?movie .
   ?movie  dbo:director  ?director .
}
```

Figure 5.5: Surface form of the predicted augmented query based on the queries in Session 2 (Figure 5.1 (b)), using classifiers that were trained on the queries of Session 1 (Figure 5.1 (a)).

```
PREFIX dbr: <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
SELECT * WHERE {
   dbo:Meryl_Streep  dbo:starring  ?movie .
   ?movie  dbo:director  ?director .
}
```

Figure 5.6: *Example query that will be prefetched by the augmented query in Figure 5.5.*

5.3.5 Augmented Queries Cache

Once the augmented query has been constructed, it is launched and evaluated on the RDF Triplestore. We store the results of the augmented queries in a cache of the form $< Q, Map < T, Set < RDF >>>$, which indicates the set of RDF triples that are results of each triple pattern T inside the query Q. When the client executes a new query, it is first checked against this cache in order to see if the query results have already been prefetched by a previous augmented query.

We do so by calculating the new query's Q-Type and checking each of its triple patterns against the cache. If all of the triple patterns are found in the cache, the cached results are then filtered in order to bind the variables introduced in the construction process of the augmented query to the resources used in the incoming query. The queries stored in the cache can be replaced using a number of cache-replacement policies, such as Least-Frequently Used (LFU) or Least-Recently Used (LRU). Given that the implementation of this cache is not part of this thesis, we do not go into further details of which cache replacement policies are most suitable for this scenario.

Example. For instance, assume that we have cached the results of the augmented query shown in Figure 5.5 and that the new query received by the SPARQL endpoint is shown in Figure 5.6. Both queries have the same Q-Type and, by comparing the triple patterns of the two queries, we can see that by binding the variable $?var_1$ to the resource $dbo : Meryl\_Streep$, we can obtain the results of the new query. Thus, we filter the cached results of the augmented query to those only containing the value $dbo : Meryl\_Streep$ and return them to the client. By doing so, we eliminate the need to execute multiple queries on the triplestore, given that the cached results of the augmented query might be relevant to several queries in the session.

Table 5.1: *Characteristics of the datasets used in our experiments. The numbers of queries and distinct queries refer to SELECT queries only. The query frequency is calculated as the number of SELECT queries received by the SPARQL endpoint per minute.*

	esDBpedia	**enDBpedia**
Total SELECT Queries	167,810	203,874
Distinct SELECT Queries	46,397	105,284
Distinct IPs	2,197	8,918
Sessions	963	619
Query Frequency per Minute	2.89	5.48
Months Covered	12	3

5.4 Approach Validation

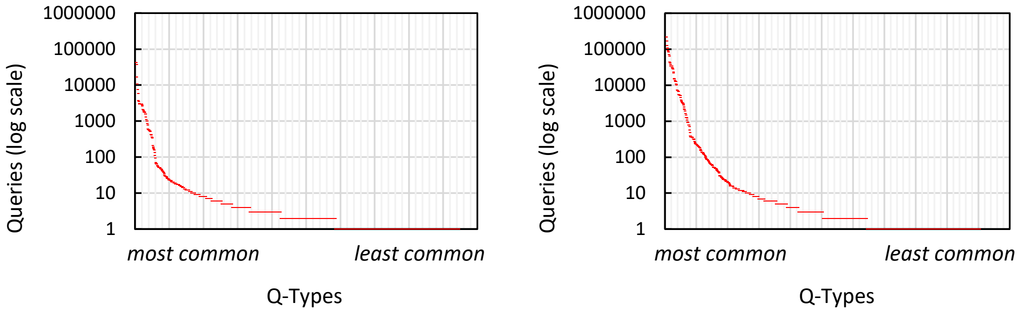
We evaluated our approach by studying the Spanish DBpedia (esDBpedia) query logs extracted directly from the esDBpedia SPARQL endpoint and the English DBpedia (enDBpedia) logs published for the 2013 USEWOD workshops[1]. The log files contain a sequence of requests received by the respective public SPARQL endpoints and cover different periods between 2012 and 2013. We extracted the SPARQL SELECT queries from other SPARQL queries and HTTP requests for use in our experiments. Table 5.1 shows the most relevant facts about the extracted datasets. As we can see, the esDBpedia dataset covers more months but the enDBpedia has a more diverse dataset, both in terms of distinct SELECT queries and IPs from which the queries were made.

We divided the log into one-hour query sessions according to the requesting IP and considered the n previous queries from the same session in our classifiers. We experimented with different values of n to see the influence of the number of considered queries on the results of the classifiers. For the esDBpedia dataset, we also included the time intervals between consecutive queries as additional classifier features. We could not do the same with the enDBpedia dataset because the published logs do not include the original timestamps of the queries, but rather round them up to the nearest hour in order to protect the users' privacy.

We implemented our classification problem using Weka 3.8.1[2] with the J48 Decission Tree, Bayesian Network, Bagging with REP Tree and Random Forest classifiers. In this section, we report the results using the J48 Decision Tree classifier as it performed slightly better than the other classifier in certain cases. In all of our experiments,

[1] 2013 USEWOD Workshop: https://eprints.soton.ac.uk/379399/

[2] Weka: https://www.cs.waikato.ac.nz/ml/index.html



(a) *esDBpedia* ***(b)*** *enDBpedia*

Figure 5.7: *Number of queries (in log scale) corresponding to each of the computed Q-Types. The x-axis ranks the Q-Types from most common (left) to least common (right).*

we used the ZeroR classifier, which predicts all instances to be of the most common class, as a baseline. We ran the training of all classifiers and the predictions on a local machine composed of an Intel Core i7-4600U processor (2.10GHz) with 8GB DRAM (DDR3) and a 256GB SSD (Samsung PM851).

The rest of this section details the results of our experiments and is divided in three parts, each answering one of the research questions we address.

5.4.1 RQ7: What is the prediction accuracy of the proposed query-log analysis approach?

Before measuring the prediction accuracy of the used classifiers, we calculated the number of generated Q-Types in both datasets. We found that the queries of the esDBpedia dataset correspond to 943 Q-Types whereas in the enDBpedia logs we found 3,139 Q-Types. Figure 5.7 shows the distribution of queries among the computed Q-Types plotted in logarithmic scale. We can see from Figure 5.7 that the distribution of Q-Types is very skewed, with a large number of Q-Types corresponding to few queries and only a handful of Q-Types corresponding to the majority of queries. Given this distribution, in the rest of the experiments we only consider the most common Q-Types that cover the vast majority of the queries. More precisely, we consider 56 Q-Types that cover 98.5% of all queries in the esDBpedia dataset, whereas in the enDBpedia dataset we consider 60 Q-Types that cover 98.1% of all queries.

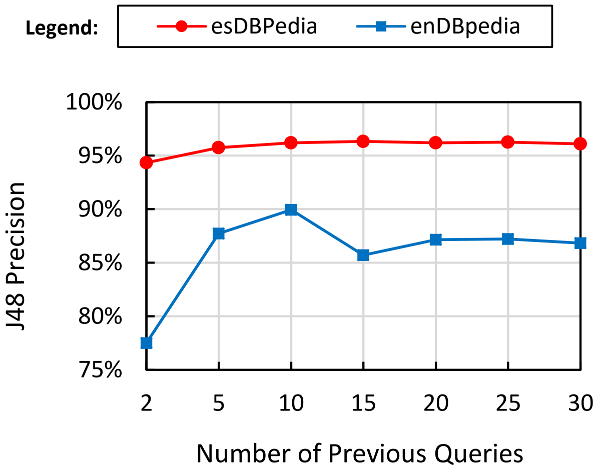


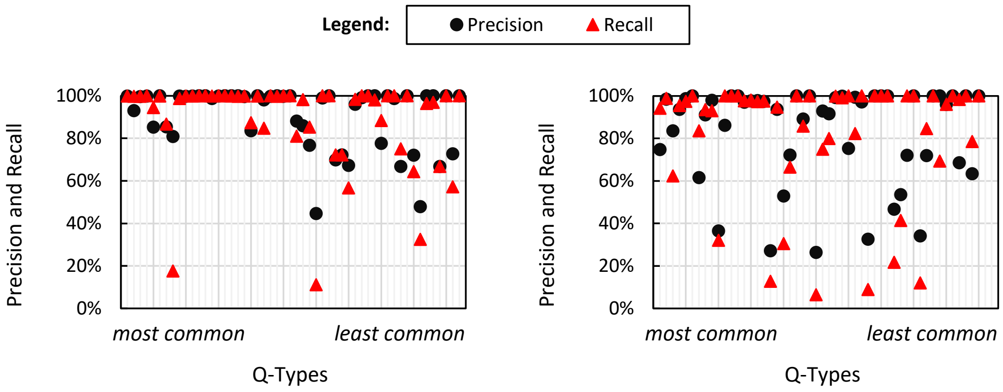
Figure 5.8: *Precision of the Q-Type classifier.*

Q-Type Prediction

Using the most common Q-Types, we evaluated the classifier's precision in predicting the Q-Type of the next query when considering different numbers of previous queries, n. Figure 5.8 shows the classifier precision on both datasets. For esDBpedia, the classifier achieves high accuracy even when $n = 2$ and reaches a peak of 96.34% when $n = 15$. As for the enDBpedia dataset, the classifier's peak precision of 89.95% is achieved when $n = 10$. In general, the classifier achieves worse precision with the enDBpedia dataset, which indicates that the queries received by the enDBpedia SPARQL endpoint are more diverse and do not follow a predictable pattern such as with esDBpedia. Note that the baseline for this experiment, obtained with the ZeroR classifier is 22.09% for esDBpedia and 15.35% for enDBpedia.

We also evaluated the accuracy of the classifier with less-common Q-Types. Figure 5.9 shows the classifier's precision (number of correctly-classified instances divided by total number of classified instances) and recall (number of correctly-classified instances divided by the total number of instances of the class) for each of the included Q-Types in both datasets. We chose the values of n that provide the highest overall precision to perform this experiment, namely $n = 15$ for esDBpedia and $n = 10$ for enDBpedia.

For the esDBpedia dataset, we can see that the classifier has a precision and recall of over 80% in the majority of cases and its recall only drops below 50% for 3 of the



(a) *esDBpedia (n = 15)* ***(b)*** *enDBpedia (n = 10)*

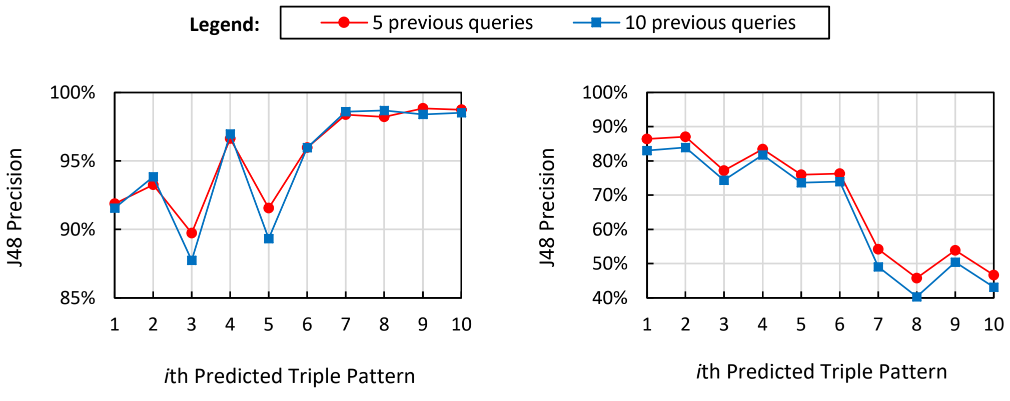
Figure 5.9: *Q-Type classifier precision and recall for each of the included Q-Types. The x-axis ranks the Q-Types from most common (left) to least common (right). Each marker represents the precision (black) or recall (orange) for a Q-Type.*

included Q-Types. On the other hand, the classifier registers a similar drop with 8 Q-Types in the case of enDBpedia. The classifier performs badly with these types because it cannot distinguish them from other types with the used features. We argue that the solution could be to include other features in the classifier models, such as the time interval between queries in enDBpedia.

Triple-Pattern Mappings Prediction

After evaluating the Q-Type prediction algorithm, we studied the accuracy of the classifiers in predicting the triple-pattern mappings (as discussed in section 5.3.2) that are used with the predicted Q-Type to construct the augmented query. Figure 5.10 shows the classifier's precision on both datasets, the x-axis indicates the number of mappings in the predicted Q-Type and the two series show the results when considering 5 and 10 previous queries. A common behavior that can be observed in figure 5.10 in both datasets is that, unlike the Q-Type classifier, increasing n does not always increase the precision of the triple-pattern classifiers. This indicates that the predicted triple patterns appear in previous queries even when $n = 5$ or $n = 10$ and any further increase only adds more unnecessary features to the classifiers model.

It is also worth noting that the classifier results are completely different among the two datasets when considering queries that have more than 6 triple patterns, with the



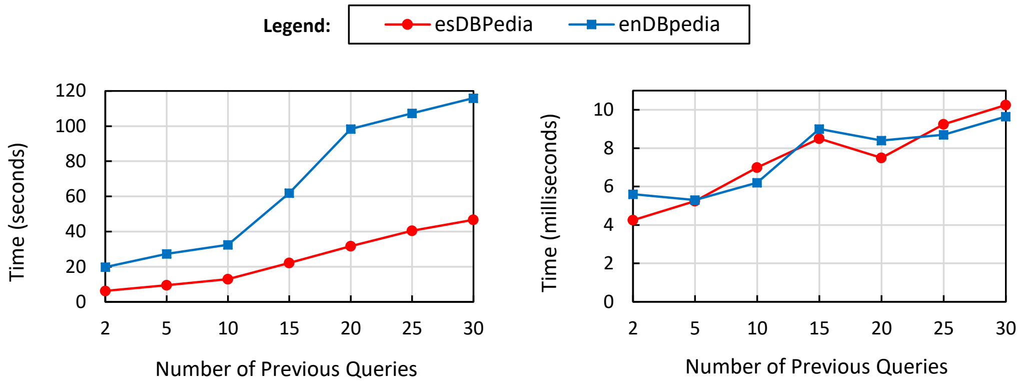
(a) *esDBpedia* ***(b)*** *enDBpedia*

Figure 5.10: *Precision of the triple patterns classifiers. For improved clarity, not all data labels are displayed on the chart.*

precision increasing to around 98% with esDBpedia and dropping to below 50% with enDBpedia. This can be explained as follows: in the esDBpedia dataset. 21.3% of queries have more than 6 triple patterns, of which 98.2% are duplicates. By contrast, the percentage of queries with more than 6 triple patterns drops to only 10.8% in the enDBpedia set, out of which only 33.7% are duplicates. The extremely high duplicates rate explains the high precision of the classifier with esDBpedia. On the other hand, the small number of queries with more than 6 triple patterns in the enDBpedia dataset, coupled with the low duplication rate, is not sufficient to train a classifier model with high precision.

5.4.2 RQ8: Can the predictions be made within a reasonable amount of time?

In order to answer this question, we measured two different execution times with the used machine learning models: **(1)** the time taken to train the machine learning models, a process that is done offline, and **(2)** the time taken to make a prediction using the trained models, which is done while the triplestore is receiving queries. We calculated these times for both the models used to predict the Q-Type as well as the ones used to predict the Triple-Pattern Mappings.



(a) *Training Time* ***(b)*** *Prediction Time*

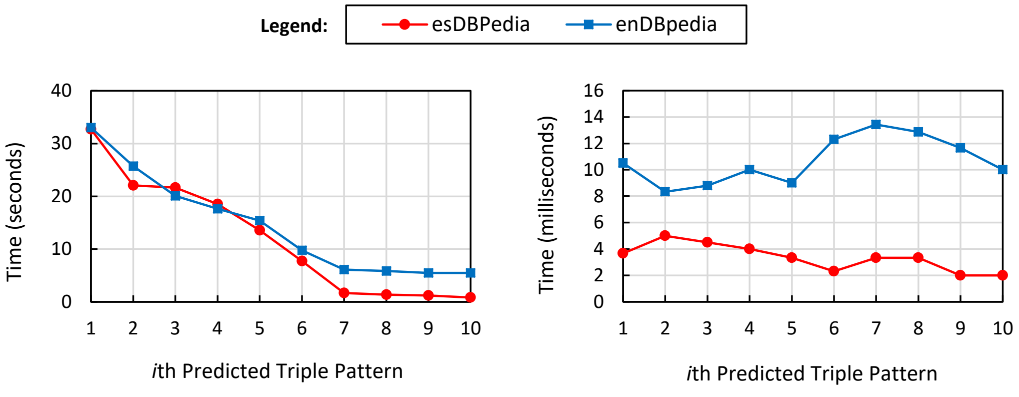
Figure 5.11: *Training and prediction times of the Q-Type classifier. The Training Time is the time taken to train the entire model (in seconds). The Prediction Time is the time taken to run 1,000 predictions through the trained model (in milliseconds).*

Q-Types

Figure 5.11 shows the training and prediction times of the Q-Type classifiers. Figure 5.11 (a) shows the time taken to train the Q-Type classifier in seconds, while Figure 5.11 (b) shows the time taken to run 1,000 Q-Type predictions through the trained model. It should be noted again here that the training process is done offline and does not affect the response times of the SPARQL endpoint.

Figure 5.11 (a) indicates that training the Q-Type classifier on the enDBpedia dataset takes considerably more than on the esDBpedia dataset, which is due to the fact that the enDBpedia dataset is more diverse and has more Q-Types. However, the longest reported training time is around 120 seconds, when using 30 previous queries, is still reasonable, given that the training is done offline.

On the other hand, Figure 5.11 (b) shows the time taken to perform 1,000 predictions using the trained Q-Type classifier. We can see from this figure that both datasets can make the predictions extremely quickly, with an average of 6 milliseconds to run 1,000 predictions. Given that the average query frequency reported in Table 5.1 is 2.89 queries per minute for esDBpedia and 5.48 queries per minute for enDBpedia, we can see that the classifier can easily predict the Q-Types of incoming queries without causing any overhead to the response times of the SPARQL endpoint.



***(a)** Training Time*

***(b)** Prediction Time*

***Figure 5.12:** Training and prediction times of the Triple-Pattern Mappings classifiers. The Training Time is the time taken to train the entire model (in seconds). The Prediction Time is the time taken to run 1,000 predictions through the trained model (in milliseconds).*

Triple-Pattern Mappings

Similarly, Figure 5.12 shows the training and prediction times of the Triple-Pattern Mappings classifier. In terms of training time, Figure 5.12 (a) shows that the time taken to train the classifiers is inversely related to the number of triple patterns in the query. This is due to the fact that most queries in both datasets have few triple patterns, which in turn results in a smaller training set of queries with a big number of triple patterns.

Figure 5.12 (a) also shows that, in general, training the Triple-Pattern Mappings classifier takes less time than the Q-Types classifier, which indicates that the complexity found in Q-Types classifier of enDBpedia is not replicated in the triple patterns. Again, this training is done offline and does not affect the query response times.

On the other hand, Figure 5.12 (b) shows the times taken to run 1,000 predictions through the trained classifier. On average, running 1,000 predictions through the Triple-Pattern Mapping classifier takes around 10 milliseconds for the enDBpedia and only 3 milliseconds for esDBpedia. This is 10^4 faster than the frequency of the queries received by the SPARQL endpoint, which indicates that predicting the Triple-Pattern Mappings using the incoming queries does not cause any delay in the query response times of the endpoint.

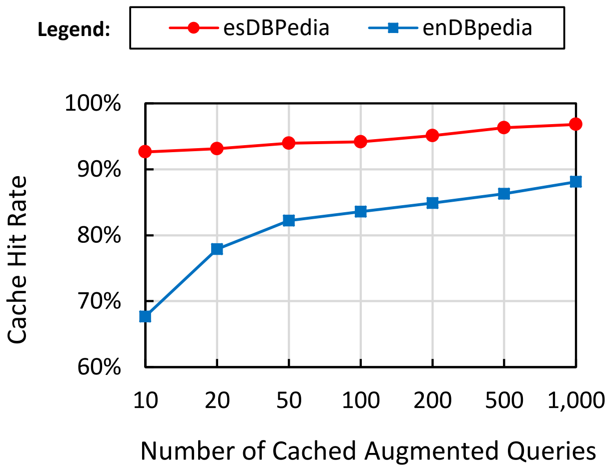


Figure 5.13: *Cache Hit Rate based on the constructed augmented queries.*

5.4.3 RQ9: What is the cache hit rate of the proposed prefetching approach?

We performed a final experiment to estimate the 'cache hit rate' that our approach can achieve by caching the results of the augmented queries predicted by our approach. While the physical cache depicted in Figure 5.2 is not fully-implemented, we estimated the 'cache hit rate' by comparing the triple patterns of the augmented queries with the triple patterns of the subsequent queries in the query session. We say that we achieve a *cache hit* when all of the triple patterns of a query have previously appeared in an augmented query executed in the same session.

Figure 5.13 shows the cache hit rates that can be achieved by caching different numbers of predicted augmented queries. It indicates that, for esDBpedia, we can have cached results for between 92.63% and 96.80% of future queries, depending on the number of cached queries. On the other hand, the hit rate for enDBpedia ranges between 67.70% when only caching 10 augmented queries and 88.10% when caching 1,000 augmented queries.

Compared to previous approaches, Zhang *et al.* reported an average cache hit rate of 76.65% using a dataset of enDBpedia queries of a similar size [84]. On the other hand, Lorey *et al.* report a cache hit rate that ranges between 54.21% when considering one-hour session to 100% for some user when caching the results of all augmented queries launched by a user within a day [55]. However, this figure is also an estimation offered by the authors without implementing this caching system and the feasibility of caching query results over such a long period is not discussed any further.

5.5 Summary

In this chapter, we presented an approach to predict data for prefetching in RDF Triplestores based on analyzing historic SPARQL query logs. More specifically, we applied a query augmentation strategy using machine learning techniques in order to prefetch data in RDF Triplestores. The approach can be applied by analyzing the query logs of the store's SPARQL endpoint and does not require any specific information to be available in the store's data graph.

Throughout the chapter we explained how we capture the similarity between SPARQL queries by constructing Query Types and Triple-Pattern Mappings. We also detailed the design of our machine learning models and how they are used to construct an augmented query that prefetches data relevant to subsequent queries in the session. Moreover, we provided the design of a full caching and prefetching system that takes advantage of our proposed approach, leaving the implementation details of system's caching component for future work.

We evaluated our approach by analyzing the SPARQL endpoint query logs of the Spanish and English DBpedia. The results indicate that the prediction of both Q-Types and Triple-Pattern Mappings does not require a large number of queries, only between 10 to 15, to achieve high precision, which indicates that our approach can be used in long as well as short sessions.

In general, the prediction accuracy is higher with the esDBpedia dataset, given the fact that the enDBpedia logs are more diverse and contain more unique queries. For a minority of cases, namely for queries containing more than 6 triple patterns, the classifier accuracy drops for the enDBpedia due to the insufficient size of this subset of queries. However, our approach can still achieve a cache hit rate of around 85% for the enDBpedia dataset, which is considerably higher than previous approaches.

Our conclusions from this chapter are that the proposed analysis of SPARQL query logs can indeed be used to generate accurate predictions that are later used to prefetch data from RDF Triplestores. By doing so, we have demonstrated the viability of our second proposed approach to analyze statements and instructions of computer languages to predict data for prefetching.

Chapter 6

Conclusions and Future Work

In this chapter, we provide a conclusion of the work done in this thesis and highlight directions for work to be carried out in the future.

6.1 Conclusions

Throughout this thesis, we have developed two approaches to data prefetching: **(1)** an approach to prefetching in Persistent Object Stores based on static code analysis of object-oriented applications, and **(2)** and approach to prefetching in RDF Triplestores based on analysis of statements of the declarative query language SPARQL. Both of the developed approaches are based on predictions made from analysis of instructions and statements of computer languages, and do not need any data from the underlying storage system. We have developed a formalization of both approaches, implemented them and demonstrated their viability and benefits with a series of extensive experiments.

First, Chapter 3 presented the first contribution of the thesis **(C1)**. The chapter detailed the static code analysis approach we developed to predict data for prefetching from POSs. The approach is based on the concept of Type Graphs, which analyze the code of an application's methods and predicts which data they access. It also includes inter-procedural analysis to move the prefetching as early as possible and hence guarantee that the prefetched objects are loaded before they are accessed. We implemented the approach using Java as an example OO language and demonstrated its viability by answering the research questions **RQ1**, **RQ2**, **RQ3** and **RQ4**, which show that the approach yields high-accuracy prediction for the majority of the studied applications, makes the predictions with enough time in advance for the predicted objects to be prefetched and can be performed within a reasonable amount of time.

Afterwards, we presented the second contribution **(C2)** in Chapter 4 by using *dataClay* as an example POS to integrate the developed static code analysis approach and prefetch data from persistent storage. The implementation includes automatic modification of application source code to enable prefetching as well as automatic loading of data to maximize the benefits obtained from prefetching the data. We evaluated the developed approach by answering the research questions **RQ5** and **RQ6** using the OO7 and Wordcount benchmarks. The obtained experimental results show that our prefetching approach results in greater reductions of application execution times, as well as higher object hit rates, than previous prefetching approaches.

Finally, we developed the last contribution **(C3)** in Chapter 5. The chapter explains an approach that analyzes the statements of SPARQL query logs and detects recurring query patterns and uses the detected patterns to construct augmented queries that prefetch data relevant to subsequent requests. We implemented our prediction approach and tested it with the real-world query logs of esDBPedia and enDBpedia to answer the research questions **RQ7**, **RQ8** and **RQ9**. The experimental results show that the approach yields high-accuracy prediction, is performed in minimal time, and thus does not add any overhead, and achieves higher cache hit rate than previous approaches.

By doing so, we have fulfilled the main aim of this thesis, which was to prove the following hypothesis:

In a technological setting where applications need access to persistent data, it is possible to perform data prefetching based on predictions made by analyzing the instructions and statements of the used computer language, without involving the underlying data storage system.

6.2 Future Work

This section presents possible lines of future work related to the contributions of this thesis. Some are issues that remained out of the scope of this thesis while others are new works that can be built on top of the contributions of the thesis.

6.2.1 Smart Cache Replacement Policies

The predictions made by the static code analysis approach presented in this thesis (Contribution **C1**), in terms of which persistent objects will be accessed, can be used for performance improvement techniques other than data prefetching.

One such technique is smart cache replacement policies [44, 46, 48]. Our static analysis approach allows us to know that an object o is accessed by a method m and another method m' after the execution of n different methods. Knowing this information, we can decide whether to keep the object o in memory after the first time it is prefetched. This also needs an estimation of the number of objects that will be accessed by the n methods executed between the two methods that access o, which we can also be obtained from our analysis. The exact number of objects that a memory cache can hold depends on the size of the cache, but using these predictions we can estimate which objects should be removed from the cache, instead of applying a simple Least-Frequently Used (LFU) or Least-Recently Used (LRU) policy.

6.2.2 Dynamic Data Placement

Another technique that can be applied using the predictions made by our static code analysis approach is dynamic data placement [53, 58]. For instance, the analysis allows us to know the set of objects that each method of the application will access when executed. Given a system similar to *dataClay*, we can use this information to distribute the objects when the application starts executing in a way that allows the sets of objects needed by different methods are located in different nodes. This in turn allows us to prefetch the objects needed for the execution of two consecutive methods in parallel, which maximizes the benefits obtained from this prefetching.

6.2.3 Additional Predictions of SPARQL Queries

The third contribution of this thesis **(C3)** currently uses the structure and triple patterns of SPARQL SELECT queries in the classifier models used to predict the augmented queries that prefetch data. These models can be extended to take into account other features of SELECT queries, such as FILTER clauses. The FILTER keyword is used in a

SELECT query to limit the values of an introduced variable to a defined set of values. Taking these FILTER clauses of previous queries into consideration when building the classifier models allows us to predict which variables of the constructed augmented queries would have a related FILTER clause. This in turn allows us to limit the values of the introduced variable to a specified set, hence reducing the number of triples prefetched by the augmented query.

Another possible extension of this contribution is to take into consideration other forms of SPARQL queries, such as ASK or DESCRIBE. For instance, ASK queries are used to test whether or not a query graph pattern has a solution. No information is returned about the possible query solutions, only *true* if a solution exists or not otherwise. Hence, predicting the solution of an ASK query based on previous queries allows us to prefetch the answer, *true* or *false*, without the need to evaluate the query graph pattern.

6.2.4 Human Query Sessions vs. Machine Query Sessions

Finally, another possible line of future work related to the contribution **C3** is distinguishing human query sessions from session made by machine agents. Previous studies show that human sessions and machine sessions exhibit different behaviour when it comes to the queries they launch [59, 62]. On the one hand, human session tend to follow a trial-and-error pattern in which consecutive queries add or remove triple patterns to previous queries with the aim of finding the needed information. On the other hand, machine sessions tend to use the same query template with a change in one or more triple patterns coming from a predefined set of values. Understanding these differences between human sessions and machine session allows to test the effectiveness of our approach on both types and optimize it accordingly.

6.2.5 Index-Only Query Answering for SPARQL Endpoints

Index-Only Query Answering (IOQA) refers to the technique of answering queries by only using indexes created on the data. Given that data in RDF Triplestores is represented through triples, we only need to create eight indexes to index all of the available data. These indexes are: subject, object, predicate, subject-object, subject-predicate, predicate-object and subject-object-predicate. For this reason, IOQA is a popular query optimization method for RDF Triplestores [37, 57].

In this thesis, we predict which Q-Types and Triple-Pattern Mappings should be used to construct an augmented query that prefetches data. However, the predictions of Triple-Pattern Mappings can also be used to decide which indexes are sufficient to answer the queries in a query session through IOQA, and create these indexes. For instance, if the Triple-Pattern Mappings indicate that the subject of the triple is the part that changes most frequently, we can use this information to create an index on the subject alone, which would allow us to answer the queries in the session using IOQA.

One limitation of IOQA is the fact that all of the created indexes should fit entirely in memory, otherwise we would have to load the index from persistent storage to answer the queries which defeats the purpose of this optimization. Thus, we can also combine our prefetching approach with IOQA by estimating the size of the indexes needed to perform IOQA and comparing it with the size of the available memory. We can then use our predictions to either perform prefetching or IOQA based on whether the needed indexes can fit entirely in memory.

6.3 Results Dissemination and Collaborations

During the course of this thesis, several collaboration were undertaken with different institutions. The collaborations, as well as the contributions presented in this thesis, produced the following publications:

- **R. Touma**, A. Queralt, M. S. Pérez and T. Cortes. "Predicting access to persistent objects through static code analysis". *New Trends in Databases and Information Systems (ADBIS'17).* pp 54–62. Springer (2017).

  This publication corresponds to the first contribution of the thesis **(C1)** and was developed at the Storage Systems Group at the Barcelona Supercomputing Center (BSC).

- **R. Touma**, A. Queralt and T. Cortes. "CAPre: Code-Analysis based Prefetching for Persistent Object Stores". *Under review in the 22nd International Conference on Extending Database Technology (EDBT'19).* Notification date: November 23rd, 2018.

  This publication corresponds to the second contribution of the thesis **(C2)** and was developed at the Storage Systems Group at the Barcelona Supercomputing Center (BSC).

- M. Rico, **R. Touma**, A. Queralt and M. S. Pérez. "Machine-Learning based query augmentation for SPARQL endpoints". *In Proceedings of the 14th International Conference on Web Information Systems and Technologies (WEBIST'18).* pp 57–67. SciTePress (2018). **Best Student Paper Award**.

  This publication corresponds to the this contribution of the thesis **(C3)** and was developed during a six-month secondment at the Ontology Engineering Group (OEG) in the Universidad Politécnica de Madrid (UPM).

- A. Kiatipis, A. Brandon, **R. Touma**, P. Matri, M. Zasadzinski, T. L. Nguyen, A. Lebre and A. Costan. "A Survey of Benchmarks to Evaluate Data Analytics for Smart-* Applications". *Under review in IEEE Transactions on Big Data - Special Issue on Edge Analytics in the Internet of Things.*

  This publication is the result of a collaboration with other PhD candidates in the BigStorage European Training Network (ETN)[1], funded by the European Commission's Horizon 2020 Marie Sklodowska-Curie Actions.

[1] BigStorage ETN: http://bigstorage-project.eu/

Bibliography

[1] Ahn, Jung-Ho and Kim, Hyoung-Joo. “Dynamic SEOF: An Adaptable Object Prefetch Policy for Object-oriented Database Systems”. In: *The Computer Journal* 43.6 (2000), pp. 524–537. eprint: http://comjnl.oxfordjournals.org/content/43/6/524.full.pdf+html.

[2] Ahn, Jung Ho and Kim, Hyoung Joo. “SEOF: an adaptable object prefetch policy for object-oriented database systems”. In: *Proceedings of 13th ICDE*. 1997, pp. 4–13.

[3] Aly, Mohamed. “Survey on multiclass classification methods”. In: *Neural networks* (2005), pp. 1–9.

[4] Atkinson, M. P. et al. “An Approach to Persistent Programming”. In: *The Computer Journal* 26.4 (1983), p. 360.

[5] Baer, Jean Loup and Chen, Tien Fu. “An Effective On-chip Preloading Scheme to Reduce Data Access Penalty”. In: *Proceedings of the 1991 SC*. ACM, 1991, pp. 176–186.

[6] Bernstein, Philip A, Pal, Shankar, and Shutt, David. “Context-based prefetch for implementing objects on relations”. In: *VLDB*. Vol. 99. 1999, pp. 7–10.

[7] Bizer, C., Heath, T., and Berners-Lee, T. “Linked data - the story so far”. In: *Int. J. Semantic Web Inf. Syst.* 5.3 (2009), 1–22.

[8] Blair, Stuart Andrew. “On the classification and evaluation of prefetching schemes”. PhD thesis. University of Glasgow, 2003.

[9] Booch, Grady. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2004.

[10] Bowman, I.T. and Salem, K. “Semantic Prefetching of Correlated Query Sequences”. In: *Proceedings of 23rd ICDE*. 2007, pp. 1284–1288.

[11] Bowman, Ivan T. and Salem, Kenneth. “Optimization of Query Streams Using Semantic Prefetching”. In: *ACM Trans. Database Syst.* 30.4 (Dec. 2005), pp. 1056–1101.

[12] Box, George and Jenkins, Gwilym. *Time Series Analysis: forecasting and control.* Oakland, CA, USA: Holden-Day, 1976.

[13] Brown, A. L. and Morrison, R. “A Generic Persistent Object Store”. In: *Software Engineering Journal* 7.2 (Mar. 1992), pp. 161–168.

[14] Byna, Surendra, Chen, Yong, and Sun, Xian-He. “Taxonomy of Data Prefetching for Multicore Processors”. English. In: *Journal of Computer Science and Technology* 24.3 (2009), pp. 405–417.

[15] Cahoon, Brendon and McKinley, Kathryn S. “Data Flow Analysis for Software Prefetching Linked Data Structures in Java”. In: *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques.* 2001, pp. 280–291.

[16] Cao, Pei et al. “A study of integrated prefetching and caching strategies”. In: *Proceedings of 1995 SIGMETRICS.* Vol. 23. 1. ACM, 1995, pp. 188–197.

[17] Carey, Michael J., DeWitt, David J., and Naughton, Jeffrey F. “The 007 Benchmark”. In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data.* SIGMOD ’93. New York, NY, USA: ACM, 1993, pp. 12–21.

[18] Chen, Shimin, Gibbons, Phillip B., and Mowry, Todd C. “Improving Index Performance Through Prefetching”. In: *Proceedings of 2001 SIGMOD PODS.* Santa Barbara, California, USA: ACM, 2001, pp. 235–246.

[19] Chen, Shimin et al. “Improving Hash Join Performance Through Prefetching”. In: *ACM Trans. Database Syst.* 32.3 (Aug. 2007).

[20] Chen, Tse-Hsun et al. “Detecting Performance Anti-patterns for Applications Developed Using Object-relational Mapping”. In: *Proceedings of the 36th International Conference on Software Engineering.* ICSE 2014. Hyderabad, India: ACM, 2014, pp. 1001–1012.

[21] Community, Hibernate. *Hibernate Documentation - Chapter 19 - Improving Performance.* URL: https://docs.jboss.org/hibernate/orm/3.3/reference/en/html/performance.html.

[22] Curewitz, Kenneth M., Krishnan, P., and Vitter, Jeffrey Scott. “Practical Prefetching via Data Compression”. In: *SIGMOD Rec.* 22.2 (1993), pp. 257–266.

[23] DataNucleus. *DataNucleus - JDO Fetch-Groups.* URL: http://www.datanucleus.org/products/accessplatform_4_1/jdo/fetchgroup.html.

[24] Dimitrov, Martin and Zhou, Huiyang. “Combining Local and Global History for High Performance Data Prefetching”. In: *In The Journal of InstructionLevel Parallelism Data Prefetching Championship*. 2009.

[25] Ding, Luping et al. “Application-Specific Schema Design for Storing Large RDF Datasets”. In: *Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems*. Vol. 89. CEUR Workshop Proceedings. CEUR-WS.org, 2003.

[26] Dividino, Renata and Gröner, Gerd. “Which of the following SPARQL Queries are Similar? Why?” In: *Proceedings of the First International Workshop on Linked Data for Information Extraction (LD4IE 2013)*. 2013, pp. 1–12.

[27] Django. *QuerySet API Reference - Django Documentation*. URL: https://docs.djangoproject.com/en/1.9/ref/models/querysets/#django.db.models.query.QuerySet.prefetch_related.

[28] Elbassuoni, Shady, Ramanath, Maya, and Weikum, Gerhard. “Query Relaxation for Entity-relationship Search”. In: *Proceedings of the 8th Extended Semantic Web Conference on The Semanic Web: Research and Applications - Volume Part II*. ESWC’11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 62–76.

[29] Fraser, Gordon and Arcuri, Andrea. “A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite”. In: *ACM Trans. Softw. Eng. Methodol.* 24.2 (Dec. 2014), 8:1–8:42.

[30] Gao, Sidan and Anyanwu, Kemafor. “PrefixSolve: Efficiently Solving Multi-source Multi-destination Path Queries on RDF Graphs by Sharing Suffix Computations”. In: *Proceedings of the 22Nd International Conference on World Wide Web*. WWW ’13. New York, NY, USA: ACM, 2013, pp. 423–434.

[31] Garbatov, Stoyan and Cachopo, João. “Data access pattern analysis and prediction for object-oriented applications”. In: *INFOCOMP Journal of Computer Science* 10.4 (2011), pp. 1–14.

[32] Gerlhof, CarstenA. and Kemper, Alfons. “A multi-threaded architecture for prefetching in object bases”. English. In: *Proceedings of the 4th EDBT*. Vol. 779. Springer Berlin Heidelberg, 1994, pp. 351–364.

[33] Gierke, Oliver et al. *Spring Data JPA - Reference Documentation*. URL: http://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.entity-graph.

[34] Han, Wook-Shin, Loh, Woong-Kee, and Whang, Kyu-Young. "Type-Level Access Pattern View: A Technique for Enhancing Prefetching Performance". In: *Proceedings of the 1th DASFAA*. Springer, 2006, pp. 389–403.

[35] Han, Wook-Shin, Moon, Yang-Sae, and Whang, Kyu-Young. "PrefetchGuide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational DBMSs". In: *Information Sciencies* 152 (2003), pp. 47–61.

[36] Han, Wook-Shin, Whang, Kyu-Young, and Moon, Yang-Sae. "A Formal Framework for Prefetching Based on the Type-Level Access Pattern in Object-Relational DBMSs". In: *IEEE Trans. Knowl. Data Eng.* 17.10 (2005), pp. 1436–1448.

[37] Harth, A. and Decker, S. "Optimized index structures for querying RDF from the Web". In: *Third Latin American Web Congress (LA-WEB'2005)*. Oct. 2005, 10 pp.–.

[38] Hartig, Olaf, Bizer, Christian, and Freytag, Johann-Christoph. "Executing SPARQL Queries over the Web of Linked Data". In: *Proceedings of the 8th International Semantic Web Conference*. ISWC '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 293–309.

[39] He, Zhen and Marquez, Alonso. "Path and cache conscious prefetching (PCCP)". In: *The VLDB journal* 16.2 (2007), pp. 235–249.

[40] Hogan, Aidan et al. "Towards Fuzzy Query-Relaxation for RDF". In: *Proceedings of the 9th Extended Semantic Web Conference, ESWC 2012*. Springer Berlin Heidelberg, 2012, pp. 687–702.

[41] Hollmann, Jochen, Ardö, Anders, and Stenström, Per. "An Evaluation of Document Prefetching in a Distributed Digital Library". In: *Proceedings of Research and Advanced Technology for Digital Libraries*. Ed. by Koch, Traugott and Sølvberg, Ingeborg Torvik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 276–287.

[42] Hurtado, Carlos A., Poulovassilis, Alexandra, and Wood, Peter T. "Query Relaxation in RDF". In: *Journal on Data Semantics X*. Springer-Verlag, 2008, pp. 31–61.

[43] Ibrahim, Ali and Cook, WilliamR. "Automatic Prefetching by Traversal Profiling in Object Persistence Architectures". English. In: *Proceedings of 2006 ECOOP*. Springer Berlin Heidelberg, 2006, pp. 50–73.

[44] Jaleel, Aamer et al. "CRUISE: Cache Replacement and Utility-aware Scheduling". In: *SIGARCH Comput. Archit. News* 40.1 (Mar. 2012), pp. 249–260.

[45] Jeon, H. Seok and Noh, Sam H. "A Database Disk Buffer Management Algorithm Based on Prefetching". In: *Proceedings of the 7th CIKM.* Bethesda, Maryland, USA: ACM, 1998, pp. 167–174.

[46] Jeong, J. and Dubois, M. "Cost-sensitive cache replacement algorithms". In: *The Ninth International Symposium on High-Performance Computer Architecture, 2003. HPCA-9 2003. Proceedings.* 2003, pp. 327–337.

[47] Karlsson, Magnus, Dahlgren, Fredrik, and Stenström, Per. "A Prefetching Technique for Irregular Accesses to Linked Data Structures". In: *Proceedings of the 6th HPCA.* IEEE, 2000, pp. 206–217.

[48] Keramidas, G., Petoumenos, P., and Kaxiras, S. "Cache replacement based on reuse-distance prediction". In: *2007 25th International Conference on Computer Design.* 2007, pp. 245–250.

[49] Kindler, Eugene and Krivy, Ivan. "Object-Oriented Simulation of systems with sophisticated control". In: *International Journal of General Systems* 40.3 (2011), pp. 313–343.

[50] Knafla, Nils. "A prefetching technique for object-oriented databases". English. In: *Advances in Databases.* Vol. 1271. Springer Berlin Heidelberg, 1997, pp. 154–168.

[51] Knafla, Nils. "Analysing object relationships to predict page access for prefetching". In: *Proceedings of the 8th POS and the 3rd PJW.* Morgan Kaufmann Publishers Inc. 1999, pp. 160–170.

[52] Knafla, Nils. "Prefetching Techniques for Client/Server, Object-Oriented Database Systems". PhD thesis. Citeseer, 1999.

[53] Lee, Chia-Wei et al. "A Dynamic Data Placement Strategy for Hadoop in Heterogeneous Environments". In: *Big Data Research* 1 (2014). Special Issue on Scalable Computing for Big Data, pp. 14 –22.

[54] Lorey, Johannes and Naumann, Felix. "Caching and Prefetching Strategies for SPARQL Queries". In: *The Semantic Web: ESWC 2013 Satellite Events: ESWC 2013 Satellite Events, Montpellier, France, May 26-30, 2013, Revised Selected Papers.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 46–65.

[55] Lorey, Johannes and Naumann, Felix. "Detecting SPARQL Query Templates for Data Prefetching". In: *The Semantic Web: Semantics and Big Data: 10th International*

Conference, ESWC 2013, Montpellier, France, May 26-30, 2013. Proceedings. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 124–139.

[56] Luk, Chi-Keung and Mowry, Todd C. “Compiler-based Prefetching for Recursive Data Structures”. In: *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS VII. New York, NY, USA: ACM, 1996, pp. 222–233.

[57] Luo, Yongming et al. “Storing and Indexing Massive RDF Datasets”. In: *Semantic Search over the Web.* Ed. by De Virgilio, Roberto, Guerra, Francesco, and Velegrakis, Yannis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 31–60.

[58] Maheshwari, Nitesh, Nanduri, Radheshyam, and Varma, Vasudeva. “Dynamic energy efficient data placement and cluster reconfiguration algorithm for MapReduce framework”. In: *Future Generation Computer Systems* 28.1 (2012), pp. 119 –127.

[59] Mario, Arias et al. “An empirical study of real-world SPARQL queries”. In: *1st International Workshop on Usage Analysis and the Web of Data USEWOD 2011.* 2011.

[60] Martí, Jonathan et al. “Dataclay: A distributed data store for effective inter-player data sharing”. In: *Journal of Systems and Software* 131 (2017), pp. 129 –145.

[61] Meng, Xiandong and Chaudhary, Vipin. “An Adaptive Data Prefetching Scheme for Biosequence Database Search on Reconfigurable Platforms”. In: *Proceedings of the 2007 ACM Symposium on Applied Computing.* SAC ’07. New York, NY, USA: ACM, 2007, pp. 140–141.

[62] Möller, Knud et al. “Learning from linked open data usage: patterns & metrics”. In: *Proceedings of the WebSci10: Extending the Frontiers of Society On-Line,* 2010.

[63] Mowry, Todd C., Lam, Monica S., and Gupta, Anoop. “Design and Evaluation of a Compiler Algorithm for Prefetching”. In: *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS V. New York, NY, USA: ACM, 1992, pp. 62–73.

[64] Neo4J. *Neo4J OGM - An Object Graph Mapping Library for Neo4j.* URL: https://neo4j.com/docs/ogm-manual/current/.

[65] Nesbit, Kyle J. and Smith, James E. “Data Cache Prefetching Using a Global History Buffer”. In: *Proceedings of the 10th HPCA.* IEEE, 2004, pp. 96–.

[66] Nilakant, Karthik et al. “PrefEdge: SSD Prefetcher for Large-Scale Graph Traversal”. In: *Proceedings of 7th SYSTOR.* Haifa, Israel: ACM, 2014, 4:1–4:12.

[67] Oren, Nir. *A Survey of Prefetching Techniques*. Technical Report CS-2000-10. Johannesburg, South Africa: University of the Witwatersand, 2000.

[68] Pan, Yue et al. "Prefetching RDF Triple Data". Pat. US 2012/013.6875 A1. IBM. 2012.

[69] Patterson, David et al. "A Case for Intelligent RAM". In: *IEEE Micro* 17.2 (Mar. 1997), pp. 34–44.

[70] Patterson, R. H. et al. "Informed Prefetching and Caching". In: *SIGOPS Oper. Syst. Rev.* 29.5 (Dec. 1995), pp. 79–95.

[71] Pérez, Jorge, Arenas, Marcelo, and Gutierrez, Claudio. "Semantics and Complexity of SPARQL". In: *ACM Trans. Database Syst.* 34.3 (Sept. 2009), 16:1–16:45.

[72] Picalausa, Francois and Vansummeren, Stijn. "What are real SPARQL queries like?" In: *Proceedings of the International Workshop on Semantic Web Information Management*. ACM. 2011, p. 7.

[73] Ramachandra, Karthik and Sudarshan, S. "Holistic Optimization by Prefetching Query Results". In: *Proceedings of the 2012 SIGMOD PODS*. Scottsdale, Arizona, USA: ACM, 2012, pp. 133–144.

[74] Sedgewick, Robert and Wayne, Kevin. *Algorithms, 4th Edition - Graphs*. [Accessed 09/10/2018]. 2016.

[75] SemanticWeb Wiki. *SPARQL Endpoint*. URL: http://semanticweb.org/wiki/SPARQL_endpoint.html.

[76] Soundararajan, Gokul, Mihailescu, Madalin, and Amza, Cristiana. "Context-aware Prefetching at the Storage Server". In: *Proceedings of the 2008 USENIX*. ATC'08. Boston, Massachusetts: USENIX Association, 2008, pp. 377–390.

[77] SourceForge. *JMCA - Java Method Cohesion Analyzer*. URL: https://sourceforge.net/projects/jmca/.

[78] Stoutchinin, Artour et al. "Speculative Prefetching of Induction Pointers". In: *Proceedings of the 10th International Conference on Compiler Construction*. Ed. by Wilhelm, Reinhard. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 289–303.

[79] VanDeBogart, Steve, Frost, Christopher, and Kohler, Eddie. "Reducing Seek Overhead with Application-directed Prefetching". In: *Proceedings of the 2009 USENIX*. San Diego, California: USENIX Association, 2009, pp. 24–24.

[80] Voelker, Geoffrey M. et al. “Implementing Cooperative Prefetching and Caching in a Globally-managed Memory System”. In: *SIGMETRICS Perform. Eval. Rev.* 26.1 (June 1998), pp. 33–43.

[81] W3C. *Resource Description Framework (RDF): Concepts and Abstract Syntax*. URL: https://www.w3.org/TR/rdf-concepts/.

[82] Wala, IBM. *Wala Wiki*. URL: http://wala.sourceforge.net/wiki/index.php/Main_Page.

[83] Wu, Zining. *Managing Multi-Tiered Non-Volatile Memory Systems for Cost and Performance*. 2016.

[84] Zhang, Wei Emma et al. “SECF: Improving SPARQL Querying Performance with Proactive Fetching and Caching”. In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. SAC ’16. New York, NY, USA: ACM, 2016, pp. 362–367.

[85] Zhu, PengFei et al. “Improving Memory Access Performance of In-Memory Key-Value Store Using Data Prefetching Techniques”. In: *Advanced Parallel Processing Technologies*. Springer, 2015, pp. 1–17.