University of Wollongong

Research Online

University of Wollongong Thesis Collection 1954-2016

University of Wollongong Thesis Collections

2015

Performance tuning of object-relational applications

Zahra Davar
University of Wollongong

Follow this and additional works at: https://ro.uow.edu.au/theses

University of Wollongong

Copyright Warning

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site.

You are reminded of the following: This work is copyright. Apart from any use permitted under the Copyright Act 1968, no part of this work may be reproduced by any process, nor may any other exclusive right be exercised, without the permission of the author. Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.

Unless otherwise indicated, the views expressed in this thesis are those of the author and do not necessarily represent the views of the University of Wollongong.

Recommended Citation

Davar, Zahra, Performance tuning of object-relational applications, Master of Computer Science - Research thesis, School of Computing and Information Technology, University of Wollongong, 2015. https://ro.uow.edu.au/theses/4598

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au

Performance Tuning of Object-Relational Applications

A thesis submitted in fulfillment of the
requirements for the award of the degree

Master of Computer Science - Research

from

UNIVERSITY OF WOLLONGONG

by

(Mina) Zahra Davar

School of Computing and Information Technology
November 2015

© Copyright 2015

by

(Mina) Zahra Davar

All Rights Reserved

Dedicated to

My mother and my father and my two beautiful angels Runika and Melody

Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

(Mina) Zahra Davar
November 26, 2015

Abstract

Optimisation of object-relational database applications implemented as a combination of object-oriented and non-procedural code requires accurate balancing of the data-processing load between the client and the server sides. The large amounts of procedural code and less efficient, overly simple algorithms, lead to the majority of data processing to be performed on the client side. As a consequence it usually increases the amounts of transmitted data to the client and the amount of time required to process the data by the client.

This thesis addresses the performance problem of object-oriented client-side applications that access data on the server through an object-oriented view of a relational database. To increase the performance of object-oriented applications, a collection of transformation rules are introduced to replace the typical iterative structures of procedural code with the equivalent structures of non-procedural code. The rules can eliminate the iterations over classes of objects on the client side and can improve the balancing of the data processing load between the client and the server. This thesis shows how object-relational database applications can be optimised by using the proposed transformation rules. The correctness of the rules is proved by the *Hoare logic* formula. Software patterns proposed in this thesis can be used for the automatic optimisation of object-relational applications.

Acknowledgement

I would like to express my sincere appreciation to the Faculty of Engineering and Information Sciences staff for all their support. I would particularly like to thank Prof. Aditya Ghose, for his help and encouragement and also A/Prof. Markus Hagenbuchner for his support and advice. This thesis would not have been possible without the ideas and the financial support from Dr. Janusz Getta.

I would like to express my deepest gratitude to the Head of Postgraduate Study of the School Prof. Minjie Zhang for her useful advice during my study and also A/Prof. Lei Wang for his support during the thesis submission process.

In particular, I would like to thank two of my friends for their special contributions. Dr. Madeleine Strong Cincotta, for her assistance and feedback and friendly comments during writing my papers and thesis. Diane Coves, for her wonderful support and positive energy from the first day of my study at this University. To these two ladies: Thank you very much.

I must thank all the members of the School of Computing and Information Technology, including the students. Special thanks to my lovely friends in our research lab, lab 239, for making such a friendly atmosphere for research and the enlightening discussions. I would like to especially thank Dr. Chao Yu(Roger), who was the most encouraging element in my research progress especially in the publishing of my first paper. Also, I would like to thank Sao Lan Leong (Marco), who is definitely one of the most motivated and intelligent people I've had the pleasure of chatting with.

I would also like to thank Hongyu Liu, Leiting Chen and Handoko, the co-authors of my papers for their contribution and teamwork during the writing of the papers.

Last but not least, to my husband: Thank you for your patience during the past two years and thank you for always being there for me.

Publications

1. Hongyu Liu, Leiting Chen, Zahra Davar, Mohammad Ramezanian Pour. Insecurity of an Efficient Privacy-preserving Public Auditing Scheme for Cloud Data Storage, Journal of Universal Computer Science, UCS, pages 473-482, 2015.

2. Zahra Davar and Handoko. Refactoring Object-Relational Database Applications by Applying Transformation Rules to Develop better Performance. In Proceedings of the 16th International Conference on Information Integration and Web-based Applications Services. ACM, pages 283-288, Hanoi Vietnam, 2014.

3. Zahra Davar , Janusz R. Getta and Handoko. Performance Optimisation of Object-Relational Database Applications in Client-Server Environments. In Proceedings of the 9th International Conference on Software Engineering Advances, ICSEA, pages 637-644, Nice France, 2014.

4. Zahra Davar and Janusz R. Getta. Performance Tuning of Object-Oriented Applications in Distributed Information Systems. In Proceedings of the 16th International Conference on Enterprise Information Systems, SCITEZPRESS Digital Library Springer Verlag, ICEIS, pages 201-208, Lisbon Portugal, 2014.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Object-relational mapping and efficient implementation of object-relational applications have received considerable attention, especially in business and commercial environments [Ors06]. The performance of object-relational applications is a serious challenge for programmers and database researchers [DD88]. Also, distributed information systems are becoming increasingly important for large organisations. At distributed system framework, query processing transfers data between computers in a network [RB13]. Therefore, performance tuning of object-relational applications in distributed information systems is considered another important challenge for research [Lop04].

Object-relational database application is a typical client/server application [SIC98]. Object-relational mapping converts relational database systems (that are available on the server side) visible to an application programmer as a collection of classes of objects on the client side. This means that relational tables on the server side are wrapped into classes on the client side, so that objects and methods can be used on the client side as well. This is why object-relational database applications are always implemented in object-oriented programming language with embedded simple non-procedural statements of the object query language (OQL).

Application programmers access data on the client side through iterations over classes of objects from the results of processing of OQL statements. They typically focus on the logic of an application rather than data processing on the server side. Such an approach for the implementation of object-relational applications tends to reduce the amount of non-procedural code and to significantly simplify the code when accessing the object-oriented view of the database. The two main reasons leading to such performance problems are as follow :

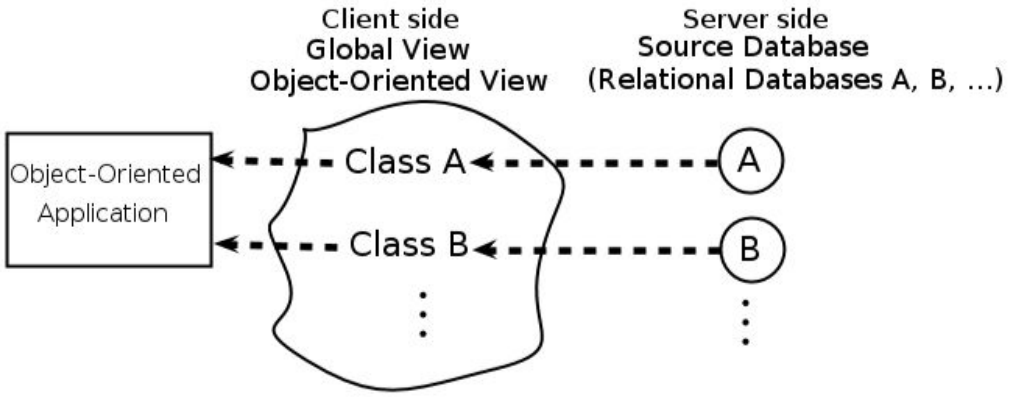

Figure 1.1: Model of the distributed system

Firstly, the iterations over the large classes of objects on the client side require a transfer of large amounts of data from the server side. This is a procedure which is inefficient for large databases. Secondly, to process this data on the client side, the application programmer uses algorithms. These algorithm are not as efficient as the ones processing the same data on the server side. For example, as mentioned in an earlier paper [DG14], a traversal of an association on the client side is typically implemented as a join of two relational tables on the server side. The implementation of the *join* operation on the server side with hash-based or index-based algorithms is much more efficient than implementation of the same *join* operation on the client side with a simple nested loop algorithm.

In a distributed system the performance problem of object-relational applications is more critical than in a client-server system. The transmission of data in the client-server system is local whereas, in the distributed system, large amounts of data are transmitted over a wide network. However, there are some similarities between the performance problem of object-relational application in these two systems. In the distributed system, there is a class of distributed information systems. This class consists of a distributed and homogeneous database system on the source side and a global object-oriented view of data on the client side. The source database side includes relational database systems and the applications on the client side accesses data from the global object-oriented view of all distributed databases.

Figure 1.1 illustrates the model of the distributed system. On the server side, there are some relational tables, such as A and B. The arrows show that the relational databases are transferred from the source database side to the client side as classes of objects so that object-oriented developers can access the data. In distributed systems, the object-oriented view of data on the global view side of the system allows application programmers to access the data through iterations over the classes of objects. Such an approach to implement user applications reduces the amount of non-procedural code required, when accessing data. Therefore, all of the data must be transferred from the source database side to an application that accesses the global view. As a result, the filtering conditions of the application will apply to all transferred data in the global view side. This leads to iterations over a large number of objects on the global view side, which is not efficient for large databases.

1.1 Research Gap

Object-oriented developers tend to write more procedural code. Using more procedural than non-procedural code in an object-oriented application, leaves more data-processing to the client. This default structure of object-oriented applications in the global view can affect the performance of both client-server and distributed systems [DG14]. At this point the performance implication in client-server systems and distributed systems, is the same.

Implementing an efficient object-relational application can be performed by changing the balance of data-processing between the client and the server side. Shifting more amounts of data-processing to the server side leads to a reduction in the transmission of a large number of data from the server (source database side) to the client (global view side). This can be done by discovering the control structure of the program and rebuilding it with more non-procedural code rather than procedural code.

1.2 Contributions of This Thesis

The solution presented in this thesis is based on the proper understanding of control structures of an application so that it can be rebuilt with more non-procedural code. By using this approach, each relational application written by a programmer can be rewritten in a way that achieves better performance. This means that only those objects which can satisfy the filtering conditions of the application are transferred from the server to the client.

To achieve this objective, first, several experiments were conducted with object-relational applications. The experiments were applied on different implementations of an application. The first version of each set of applications were based on what usually an object-oriented programmer uses with more procedural code rather than non-procedural code. The second version of the applications was based on using more non-procedural code. The performance of both versions of the applications in each of the sets, were recorded and compared.

The transformation rules, transform non-optimised version (input component) of the object-relational application into the optimised one (output component). By using transformation rules, procedural code from the input component is replaced with non-procedural statements in the output component. This centralizes most of the data processing on the server side. We considered three categories for transformation rules: *Filtering*, *traversal of association* and *aggregation*. The axioms and rules of the Hoare logic are used to prove the correctness of the rules. The problem of covering all possible cases of input components is solved by proposing certain software patterns. These patterns make the analysis of the source code easier. The combinations of proposed patterns can be used for complex applications.

This thesis, presents a set of transformation rules which can eliminate the iteration over a large number of objects. It speeds up the performance of the application by changing the control structures of an application. By applying the rules, some of the procedural components are replaced with non-procedural statements. As a result, a faster and more efficient performance of the application is achieved.

The significance of this thesis is that the proposed method improves some commonly used server-side operations thus reducing the server side load and response times. The adopted approach is very different to the existing approaches which commonly aim at improving efficiency through de-centralization (i.e. by moving some code to the client side) as in [Aga95] [RWD01].

The presented methodology supports most standard SQL/OQL functions such as Boolean expression (AND, OR) and comparison functions. The rules can be implemented in applications with a corresponding function which returns a value. The templates presented in Chapter 5 allow a more general class of predicates to enable the use of these rules.

The remainder of this thesis is organised as follows:

Chapter 2 presents background information on object-relational applications. In Chapter 3, the transformation rules are presented. Chapter 4 proves the correctness of the rules. Chapter 5 presents the software templates of the rules. Experimental results are presented in Chapter 6. Finally, Chapter 7 provides concluding remarks.

Chapter 2

Background

2.1 Overview

In this section, different types of databases related to the topic of the thesis are presented. The relational database and relational database model were discussed and compared. Also, the OO model is discussed as a OO programming model which can merge with the relational model. Advantages and disadvantages of this combination are also discussed.

2.2 Relational Databases and Relational Database Model

Relational database management systems have been used in industry since the 1980s [CB01]. E. F. Codd, an IBM researcher, proposed the idea of a relational database model which became the basis for the implementation of relational database management systems. In the 1980s, the relational database systems revolutionised database management systems.

In relational databases, data is stored into two dimensional tables in rows and columns. The relationships among the data can be seen, by comparing the values stored in these tables. Structured query language (SQL) is used to retrieve and to manipulate data in the relational database model [MKF+03].

The tables available in OR mapping would be visible to OO programs as classes of objects. Therefore the access and retrieval of data needs iterations over the classes of objects. The normalisation of relational tables eliminates redundancy and this

leads to the distribution of the data over many relational tables [Nic07]. Therefore in order to search these tables, multiple joins of tables are necessary. This process is time-consuming and a cause for a decrease in performance [DU04].

2.3 The OO Model

In the OO model(also known as OO programming model), objects are structures which combine related data and code (operations). Each object is considered as an instance of a class. Encapsulation, inheritance and polymorphism are the properties of the OO model. Encapsulation means that code and data are grouped together to construct objects. The construction of each object can be hidden from the rest of the program. This can help developers to manage the program's structure in a simple way. An inheritance feature allows the class to inherit many of its features from other existing classes. This can make OO program easier to change [Nic07]. Polymorphism refers to operations with the same name but have a slightly different meaning when they are applied to different data. In an OO model, pointers and nesting are used to establish relationships between objects. An OO model is based on the combination of data and code, different data types and also hierarchical relationships within data types and references [DD88].

2.4 The reasons for combining Relational Database Model and OO Model

It is stated in [PRBV90] that ' A relational database management system and an OO language can be combined to yield a surprisingly effective OO database system for many applications'. However combining them can cause difficulties. This will be discussed in Section 2.5. Relational database systems and OO programming languages have complementary strengths. The main reason for combining these two systems are, on one hand, OO programming language is used by most of the developers to implement their applications and on the other hand, relational databases are mostly used to store data. The combination of these two systems can manage large amounts of data.

The more complex the collection of information and the more levels of hierarchy and cross relationships, the more difficult it is to represent it within a simple table structure of a relational database [Blo03].

It is difficult to store and manipulate complex data objects like nested objects or unstructured data with the relational database [LKK00] [SKS10]. Compared to OO programming language, they can manage these complex relationships among objects [SKS10]. This can be considered as an advantage of using OO model.

OO programming language, provides almost no support for data persistence and retrieval [Nie89]. In contrast, the relational database system provides more complex persistence of data structures. In other words, in OO programming, files are the only way to make objects persistent while the relational model uses tables and the query languages for persistence on relational tables. But these are not available in OO languages. Moreover, relational database systems can manage large amounts of data [DD88].

The relational model and the OO model are fundamentally different and combining them can be difficult. On one side there is the OO model and on the database side there is a relational model. It is very expensive to convert the relational system into the OO system. This is discussed in Section 2.5.

2.5 Problem of Combining of the Relational Model and the OO model

In order to implement an OO database application, objects are constructed at the run time of an application not at implementation time. To achieve this, functions must be written for each class which can retrieve data and store it in the database. Other functions must be written to store objects and to retrieve their data from the database. A separate table must be created for each level in the hierarchy to store hierarchical structures in a relational database [DD88]. This process is very

time-consuming and has a negative effect on the performance of the application. Furthermore, relational databases do not support the encapsulation and polymorphism as featured in the OO model. This also has a negative effect on the performance of the application.

2.5.1 Impedance Mismatch

With the OO programming languages, like Java, data is represented as interconnected graphs of objects. Relational database management systems represent data in a table-like format. When loading and storing linked structures of objects using the relational database, we came across a mismatch problem. This mismatch between the OO model and the relational model is called OR impedance mismatch. In other words, the OO model and the relational model do not have the same data structure [ICK09] [Fin01].

The definition of impedance mismatch in Wikipedia is described as "The OR impedance mismatch is a set of conceptual and technical difficulties that are often encountered when a relational database management system (RDBMS) is being used by a program written in an OO programming language or style; particularly when objects or class definitions are mapped in a straightforward way to database tables or relational schema."

In OO model we have objects with various levels of granularity. Granularity refers to the level of hierarchy of objects. In a relational model, granularity is limited to the following levels: Tables, rows, values and columns. The first mismatch between the OO model and the relational model is the result of storing objects with various levels of granularity. These objects are limited in granularity which can cause a mismatch. The second mismatch is an inheritance mismatch which is the result of the mapping of object inheritance. In Java, an object's features can be inherited from another object while in the relational database there is no support of inheritance. Therefore, the class hierarchy needs to be translated to a database schema. Association mismatch is another type of mismatch between the OO model and the relational model. Associations in OO models is achieved by object references which are directional, whereas, foreign keys are used to associate two entities in the relational model. An

association implemented as a foreign key is not directional. This means that, if an association between objects should be navigable in both directions, the associations must be found twice, once in each of the association classes.

Another mismatch is data navigation mismatch. In Java we navigate from one object to another object in the object graph. Therefore, to find a particular item we navigate from one object to another, following the associations between the objects until we reach that particular item. So the way that we access data in Java is fundamentally different than in the relational database.

Impedance mismatch commonly occurs when a developer needs access to existing data which is held in a relational database. In this situation OR mapping is one of the useful ways to prevent impedance mismatch.

2.6 Solutions for Impedance Mismatch

There are a number of solutions for impedance mismatch. These include using the OO database system, OR database systems or modifying OR mapping. In this section, each solution is discussed in detail.

2.6.1 OO Database Systems

The first solution is to use the OO database system. It transforms the relational database system to the OO database system. In [Sik03] it is claimed that "OO database systems evolved from a need to satisfy the demand for a more appropriate representation and modelling of real world entities. So OO databases provide a much richer data model than conventional (relational) databases". The problem of transforming rows into objects in the relational tables, occurs when developers need to use relational databases instead of OO databases in OO programming language [Kel97].

The data in an OO database is persistent data and when this data is read by an application, it stores data as transient data. It can also be stored as persistent data as well. As soon as the program finishes, transient data will be lost. Persistent data

can be stored and it can be reused for the next execution of the program. Persistent data can be shared, accessed and updated by a database application [KA90]. One of the advantages of the OO database systems is that there is no need for developers to maintain the relationship between persistent objects and transient objects. Another advantage of using the OO database is that the problem of impedance mismatch is solved by using a unified view of data in OO programming language and in the database. The OO database system can also support inheritance [Sta98].

OO databases have some disadvantages. They have limited support for consistency constraints and limited performance-tuning capabilities [Sik03]. Also, the problem of impedance mismatch is eliminated when the relational database is copied into the OO database. However, by using this approach, the relational database is copied into the OO database but transferring data between the relational database and OR database is very time-consuming. Therefore, without considering the performance problem of the application, this approach can solve the impedance mismatch problem.

2.6.2 The OR Database models

The OR model is a model between the relational model and the OO model. Objects in an OR database model need persistence; a data model; a query language to manipulate, retrieve and store data; and also a database model [Nie89]. In some models, the OR model uses the object view to access relational data. An object view is a way to access relational data using the OR features. It allows developers to develop OO applications without changing the underlying relational schema [Dat08].

An OR database consists of tables of objects including properties and relationships which can be manipulated by methods, stored procedures or query languages. OR database technology is a hybrid between the relational database models and OO database models. An OR database models, uses OR database management systems (ORDBMS) for data management[Nic07]. Relational databases are unable to store complex data whereas an ORDBMS can deal with large amounts of high complexity data.

One of the advantages of using an OR database model rather than the relational database for OO applications is such that it solves the problem of impedance mismatch between the OO application and the relational model [Nic07] [CB01] [ABD+89]. The OR database model is using object tables instead of relational tables to store data in an OO application. This means less programming code is required followed by less development time. Other advantages of OR databases are reusability, maintainability, flexibility and functionality.

2.6.3 OR Mapping

If a row needs to be stored in or retrieved from a relational database using an OO programming language, an object view of this row in OO language must be provided. The differences between the OO model and the relational model must be overcome [Kel97].

OR mapping allows the conversion of data between inconsistent systems like tables and objects in the OO programming languages. OR mapping is one of the most common data access techniques to cope with the problem of OR impedance mismatch. OR mapping can simplify the development of application by automating the object-to-table and table-to-object exchange. This can make the implementation of application simple.

Complex objects and relationships in the application, causes a higher level of impedance mismatch. The higher level of impedance mismatch leads to the development of additional code to copy objects into relational tables, resulting in slower application development and weak performance of the application [DU04].

2.7 Modern OR Mapping Frameworks

There are different persistent frameworks and OR Mapping(ORM) frameworks available. In this section, a survey of LINQ, Hibernate and Ruby on Rails is presented. Also, the methods are compared and analysed.

2.7.1 LINQ

The Microsoft Language Integrated Query (LINQ) is an OR mapping framework which helps developers query data. There are five different implementations for LINQ presented by Microsoft. The LINQ to SQL and LINQ to Entities are the ones that need access to relational databases using OR mapping [MEW08]. Both LINQ to SQL and LINQ to Entities ORM frameworks, support the mapping of a Microsoft SQL Server database to .NET classes [MEW08]. It executes the output of the query using LINQ.

Both of these LINQ implantations, cover OR mapping within Microsoft SQL Server and third party databases. This thesis considers mapping between the Oracle database and the Netbeans classes. Therefore, LINQ is not considered as a framework for OR mapping in this thesis.

2.7.2 Ruby on Rail

Ruby on Rails is an open-source web framework which presents different ORM tools. The most popular ORM tools within Ruby on Rails are Datamapper and Sequel [BFF+15].

Using Activerecord leads to work with models rather than data. It automatically converts data to models. Therefore, all the associations are built and based on the models. The disadvantage of Activerecord is that it increases the complexity of the objects. Therefore the run time increases. A Sequel framework is implemented based on Activerecord. It supports stored procedures and advanced database functions. Datamapper was presented as another ORM framework to complete the Activerecord and make the ORM faster. Datamapper separates the OO model from database logics.

All the presented ORM tools across Ruby on Rail are able to optimise the ORM process. However, it can also cause performance delay during the execution of the application.

2.7.3 Hibernate

Hibernates is a powerful java persistent mapping tool for OR applications which enables accessing complex data [OLM14]. It supports mapping from java classes to database tables and from java data types to SQL data types. Hibernate focuses on data-processing using JDBC and SQL. It also enables data and tables to be persistent. It uses data persistence from the relational database. Persistence simply keeps some data out of the scope of the application process [BK06]. Hibernate uses a two-layer caching system for performance solutions.

Hibernate can be used as a tool which can provide automate mapping [OLM14]. Therefore in future work, hibernate can be considered within the transformation process. This will be discussed further in Chapter 7.

2.7.4 Conclusion

Although using ORM framework can speeds up the development time of an application, it adds overhead to the application. This means that by using ORM framework memory and CPU usage will increase. Also, using ORM framework for implementing queries with simple functions such as update, delete and comparison are slower than using SQL. For these reasons, ORM frameworks are not considered as an ORM solution within this thesis.

2.8 Previous works

A number of approaches have been proposed in order to improve the efficiency of OR applications. These include performance tuning, transformation methodology, integrating data and refactoring.

2.8.1 Performance Tuning and Transformation Methods

Agarwal proposed the idea of using a client-side object cache in order to increase the performance of the application. It is suggested that the actual performance was greatly dependent on the degree to which the application can take advantage of data stored in the object cache. The problem of this method, however, is that special

care must be given to the complexity of the query so that it can return instances of commonly used classes with minimum use of joins [Aga95].

In 2006, research focused on comparing the performance of object databases and OR mapping tools. This research discussed OR mapping in open source applications [vZKB06]. This approach, however, only dealt with one framework. Moreover it was not applied on the distributed (client-server) or multi-user systems which are often used by developers.

Kalantari et al. compared the performance of the object and OR database models. Certain factors were suggested in this research and system developers need to consider them when selecting a database management system for persisting objects [KB10]. This research is based on basic query implementation which means that it did not consider complex queries.

Rahayu et al. discussed the performance evaluation of OR transformation methodology. The aim of this research was to clarify the efficiency of the operations on relational tables based on certain OR transformation methodology [RWD01]. The performance of OR transformation methodology was also compared with that of the conventional relational model. This research, however, did not compare the existing relational database design methodology and their OR transformation methodology.

Meng et al. proposed the transformation rules for the OO database systems. The rules used in this research were designed to transform the structural part of a OO database schema to an equivalent relational schema [MYK+93]. These rules provided a relational view of the OO database schema for relational users. The research is limited to the structure of a relational front end for OO database systems.

The idea of translating queries from an SQL into an OQL in an automatic way, were suggested by Mostefaoui et al. This method was based on graph representations [MK98]. A formal approach to translation of OO database queries into equivalent relational queries was proposed by Yu et al who used the same method as Mostefaoui et al. in [MK98] [YZM+95]. These works, however, did not consider all the possible forms of SQL queries. In addition, the methods suggested were not general enough to be extended to other clauses and they could not address the performance problem

of OR applications.

Grust et al (2009), developed the FERRY language designed as an intermediate language. The FERRY language permits developers to access database tables using their programming language's own syntax and idiom [TMJT09]. In 2010 the same authors extended this approach by proposing the FERRY-based LINQ to SQL approach [SBG+10]. Both proposals were based on compiling the first-order functional programs into SQL which is not an applicable approach in industry.

Recently, Chen et al. proposed a framework which can detect and prioritise instances of OR mapping performance anti-patterns [CSJ+14] and therefore improve the system's response time. This is a useful framework but this approach can only detect performance bugs and leaves the debugging process for the developer.

2.8.2 Integrating Data and Refactoring

In 2001, a pragmatic approach to schema integration was proposed by Lawrence et al. In this approach, the query processor translates semantic queries into structural expressions [LB01]. This approach needs to be standardised in order to be performed automatically.

The approach of processing the integration of complex databases using the IIS*Case was proposed in 2006 by Ivan et al. The functionality of IIS*Case needs to be extended, however, in order to be able to support the complete development of an information system [LRMP06]. In this research, the performance problem of distributed information systems was not considered as a challenge in integrating the data models.

In 2010, the idea of refactoring SQL was presented by Jacobs et al. Their method relied on finding common anti-patterns and techniques [Jac14]. The proposed method was not extensible, however, and not general enough to cover most common queries.

Another research on refactoring was done in 2006 by Michael et al. This approach [MGB10] covers the ability to reduce code volume and the number of modules and

files needed during maintenance [MGB10]. The main problem with this approach, is a reduction in performance of the application. This approach is also unable to manage the refactoring of the application automatically.

Zibran et al. proposed a model for refactoring code clones in the OO source code [ZR11b]. This model is useful for estimating the efforts required for code clone refactoring [ZR11a] but it is not applicable for industrial software systems written in other programming languages. Also, a refactoring tool suggested by Zibran et al. was the clone detection part of a refactoring system [ZR11b]. This research did not consider the performance of the output application in the implementation of their refactoring system.

In 2009, Liu et al analysed the relationships among different approaches to software refactoring [LYN+09]. In this research, however there is no support for resolution orders of software refactoring either for software development.

2.9 Conclusions

There have been many attempts to combine relational and OO concepts in order to improve performance of the applications. It has been done primarily by designing rules for the transformation from an OO model into a relational model or vice-versa. Other methods of refactoring were proposed by only a few.

Most of the current transformation methodologies are based on using first-order functional programming languages and this is not applicable in industry. Most of the current refactoring methods have a negative impact on the performance of the application.

Multiple unnecessary iterations are a main remaining problem in all of the current approaches. This means that an approach is required which can eliminate these iterations in order to achieve better efficiency.

Chapter 3

Transformation Rules

3.1 Introduction

The transformation rules optimise non-optimised versions of the OR database applications to provide the necessary efficiency. The rules replace fragments of procedural code with non-procedural statements of OQL which leave most of the data processing on the server side. The transformation rules consist of two components: input and output components. The input component is a code based on a non-optimised version of the application written by the OO programmer. The rules are applied to the input component and as a result the output is achieved. The output component is a code based on the optimised version of the application after applying the rules.The output component is achieved by using more non-procedural code and changing the structure of the input component.

This thesis assumes that OR applications consist of three categories as follows:

1. Filtering
2. Traversal of Association
    - Join Traversal
    - Anti-Join Traversal
3. Aggregation

The filtering category refers to applications which filter objects using specific conditions. The view of the OO programmer about OR database is classes of objects with associations and properties. From a programmer's point of view, the data manipulation on relational tables are hidden behind the operations on the classes and

associations. Therefore, to obtain information from more than one class, navigating through associations from one class to another class is required. This forces the application to use mapping. Both *Join Traversal* and *Anti-Join Traversal* categories, refer to navigation through associations of two classes. *Anti-Join Traversal* category is supposed to find objects from one class which are not related to the objects in the other class. For both *join traversal* and *anti-join traversal*, navigating through associations is required. The last category refers to the applications with aggregation functions.

For the rest of this chapter, *relational conditions* referred to as the conditions which are discussed are needed to navigate through the associations of two classes of objects. The *non-relational* conditions are the conditions of one class without having any relation to the other classes.

Large OR applications are a combination of different applications from the proposed categories (filtering, traversal of association and aggregation). Therefore, the rules can convert most of the applications from the non-optimised versions to an optimised one. An explanation as to how to apply the rules to the combination of the proposed categories (filtering, traversal of association and aggregation) is offered in Chapter 4.

The transformation rules support most standard SQL/OQL functions within OR applications. It can supports functions that return value such as boolean expression(AND, OR) and comparison functions.

3.2 Filtering Rule

3.2.1 Input Component

Every OR application may include an iteration over one class of objects. These applications have some filtering conditions used to select some desired objects for output. The input and output components are presented as a pair of algorithms. By using the transformation rule for the applications in the filtering category, the structure of the application is changed from a program with one *SELECT* statement

of object query language(OQL) and one *IF* clause (as shown in Algorithm 1), to a program with one *SELECT* statement and one *WHERE* clause (as shown in Algorithm 2). In Algorithm 1, t is considered as an object variable. Algorithm 1, is the input component for this rule while Algorithm 2, is the output component after applying the above changes to the input component. Also, the *"P r o c e s s i n g (t)"* is considered as a block of code that can processes an object t. The application developer can write the *"P r o c e s s i n g"* section differently depending on the aim of the application.

Assume that $\varphi\ [t.t_1, t.t_2, ..., t.t_n]$ is a filtering condition. φ is a Boolean expression. An example of $\varphi\ [t.t_1, t.t_2, ..., t.t_n]$ is: Class1. Member$_i\alpha$ Value. α is an operation such as $<$, $>$,$=$, $!=$. t_1, t_2,...,t_n all are the properties of Class1. "Value" is any number or string. The operation between the conditions can be *And*, *Or* or *Not*. The condition is based on the object t. Each filtering condition is applied by making reference to the object variable.

For instance: t.Department_name ="finance". In this example, "t" is an object from Class1 and "Department_name" is a property of Class1. The "α" in this example is "=" and "Value" is a string: "finance".

Algorithm 1: Input component

Iteration over one class of objects

```
1 for each t in (SELECT * FROM Class) do
2     if φ [t.t1, t.t2, ..., t.tn] then
3         P r o c e s s i n g (t)
```

In Algorithm 1, all of the objects from the *Class* are selected and then the filtering conditions are applied to the objects. This way of implementing the filtering part of the application leads to an iteration over all objects in *Class*. $\varphi\ [t.t_1, t.t_2, ..., t.t_n]$ shows different filtering conditions for different objects in the class. This means that each object can be filtered by a condition and those conditions can be different to each other.

Evaluation of each condition requires transmission of all objects from the database over the network. Conditions are computed on the client side which leads to having

inefficient applications due to the transmission of a large number of objects.
As an example of the input component of this rule, a pseudo-code with OQL statements is presented in Algorithm 2.

Algorithm 2: Example 1/Non-Optimised version

```
1 for each t in (SELECT * FROM Department) do
2     if t.Department_id = 123 ∧t.Department_name = "finance"then
3         Write t
```

3.2.2 Output Component

Algorithm 3 is the output component of the filtering rule. Condition in the input component $\varphi\ [t.t_1, t.t_2, ..., t.t_n]$ is converted to a OQL condition $\varphi\ [t_1, t_2, ..., t_n]$ in the output component. In this regard, the references to t has to be removed and the output component includes only the name of the properties. The following algorithm is considered as an output component for filtering applications and has the same output as *Algorithm 1.* In Algorithm 3, it is assumed that $\varphi\ [t_1, t_2, ..., t_n]$ is OQL filtering condition which filters the output.

Algorithm 3: Output component
Filtering

```
1 for each t in (SELECT * FROM Class WHERE φ [t_1, t_2, ..., t_n] ) do
2     Processing(t)
```

The filtering rule can improve the performance of the filtering applications by reducing the transmission of all objects over the network. This rule helps to shift the computation of the conditions to the database (server) side. Therefore, the data-processing is faster because the database server is able to use indexing which can eliminate iterating over all the objects.

Algorithm 4 is as an example of the output component of the filtering rule.

Algorithm 4: Example 1/Optimise version

```
1 for each k in (SELECT * FROM Department WHERE
2 Department.Department_id = 123 ∧t.Department_name = "finance")do
3 |  Write k
4
```

Both algorithm 2 and algorithm 4, return objects from the *Department* class which satisfy the filtering condition. *Department_id = 123*. 'k' is an object variable which contains the output.

3.3 Traversal of Association

3.3.1 Join Traversal

Input Component

In this section the input component of the join traversal is presented. The input component which is presented as Algorithm 5, is based on nested loops. The nested loops implementation of *join*, takes an object variable from the outer loop. The property value of this object will pass to the condition of the other loop as the object on the other side of the association. Algorithm 5 includes two nested *SELECT* statements which perform the join traversal of associations. This traversal of the association is implemented either as nested loops or as an OQL's join.

The two *SELECT* statements in this algorithm compute the *JOIN* operation. Non-relational conditions are the conditions belonging to only one class without having relation to the other class. Non-relational conditions for *Class 1* are presented in the outer *SELECT* statement as $\varphi\ [t_1, ..., t_n]$. The inner class, *Class 2*, includes both the non-relational and relational conditions. The non-relational conditions of *Class 2* are presented by $\gamma\ [s_1, s_2, ..., s_n]$. The internal structure of this condition is such that it includes different non-relational conditions for different members of *Class 2*. The operation between the conditions can be *And*, *Or* or *Not* which is different depending on different application. The relational conditions between *Class 1* and *Class 2* are denoted as γ' $[< s.s_1, t.t_1 >, ..., < s.s_n, t.t_n >]$. In this condition,

t and s are object variables which return the objects to satisfy the conditions in *Class 1* and *Class 2*. For instance, γ' $[< s.s_1, t.t_1 >]$ includes a part from class1, t.t_1, and a part from *Class 2*, s.s_1. Therefore there is relation between these two parts and the objects which satisfy the condition between these two parts would be the output. An example of φ $[t]$: φ $[t_1, ..., t_n]$ is Class1.Member$_i$=Value. An example of γ $[s]$: γ $[s_1, s_2, ..., s_n]$ is Class2.Member$_j$=Value. Value can be any type i.e. of number or string. An example of the function: γ' $[< s.s_1, t.t_1 >, ..., < s.s_n, t.t_n >]$ is Class2.Member$_i$ = Class1.Member$_j$.

The computation of the above algorithm describes in the *WHERE* clause of the inner *SELECT* statement, before concatenation, there is a non-relational statement, γ $[s_1, s_2, ..., s_n]$, which is and-ed with relational conditions. The relational conditions includes properties of both classes which are appended to OQL string. This string uses references to variable t as follow: γ' $[< s.s_1, t.t_1 >, ..., < s.s_n, t.t_n >]$. t is the object variable used for the first class and s is used for second class.

Assume that $\wedge$ means AND. In Algorithm 5, $\wedge$ is is used between non-relational conditions and relational conditions of the inner loop.

Algorithm 5: Input component

Iterations over two classes of objects

```
1  for each t in (SELECT * FROM Class1 WHERE φ [t_1, ..., t_n]) do
2      for each s in (SELECT * FROM Class2 WHERE
3      γ [s_1, s_2, ..., s_n] ∧ γ' [< s.s_1, t.t_1 >, ..., < s.s_n, t.t_n >]) do
4          P r o c e s s i n g(t,s)
```

Algorithm 6 is presented as pseudo-code with OQL statements as an example of the input component of this rule. Variables t and s are considered as object variables from the Department and the Employee respectively. The Department and Employee classes are mapped to the Department table and Employee table respectively. The variable p is a set which includes the result of t and s: p <t,s>. p <t,s>, and includes both t and s.

Algorithm 6: Example 2/Non-Optimise version

```
1 for each t in (SELECT * FROM Department) do
2     for each s in (SELECT * FROM Employee WHERE
3     t.Department_id = s. Department_id) do
4         Write p <t,s>
```

Output Component

The output component implements join in OQL. To write the output component of this rule, more non-procedural code is used than procedural. Two *SELECT* statements in the input component are merged into one *SELECT* statement with a *Join* clause to create the output component. Non-relational conditions have the same format as the input component. The relational conditions in the output component do not have references to the objects. In the output component, AND or propositional conjunction are used to merge the non-relational conditions of both classes. By using this rule, filtering conditions are applied to the objects on the server side and as a result, only the objects which satisfy the *JOIN* condition will transfer to the client side. Therefore, unnecessary data is not transmitted.

Assume that γ' $[< t_1, s_1 >, < t_1, s_2 >, ..., < t_n, s_n >]$ is a *join* condition. The entire OQL statement are transformed into a statement in SQL on the database side at the output component. The transformed statement is a join statement. Therefore, for each $< t_i, s_j >$, γ' $[< t_i, s_j >]$ is the *join* condition between the $member_i$ of class1 and the $member_j$ of class2.

Algorithm 7: Output component

Join two classes of objects

```
1 for each p in (SELECT * FROM Class1 Join Class2 on
2 γ' [< t1, s1 >, < t1, s2 >, ..., < tn, sn >]
3 WHERE
4 φ [t1, ..., tn] ∧ γ [s1, s2, ..., sn] do
5     Processing(p)
```

By using this rule, implementation of the traversal of the associations is converted

from two nested loops into a *join* in OQL. This is implemented by having relational conditions of two classes as a *join* condition. The other non-relational conditions remain as WHERE clause conditions. The relational conditions are used as *join* conditions, so the references to the object variables t and s which are used in the input component are eliminated.

Most of the data processing is shifted to the server side by using this rule. This will decrease the unnecessary transmission of data from the server side. Also, the output component is implemented in a way that allows the database system to use a *join* algorithm. It will be more efficient than the nested loops implementation in the input component. This leads to achieving a better performance of the application.

Algorithm 8: Example 2/Optimised version

```
1 for each p in (SELECT * FROM Department join Employee on
2 Department.Department_id = Employee. Department_id) do
3 Write p
```

Algorithm 8 is presented as an example of the output component of this rule. The query optimiser is allowed to find better implementation of join to retrieve data, by using join in this version of the program. Both optimise and non-optimise versions of the application results in the same output. All the objects from the Department and Employee classes which have the same Department_id are selected as the output in the presented application in Algorithm 6 and 8. Also all the object which satisfy the condition in the WHERE clause are added to the output.

3.3.2 Anti-Join Traversal

Input Component 1

Anti-join applications, retrieve the objects from one class where there is no related object in the other class. One of the frequently used implementations of *anti-join* is introduced as Algorithm 9. The input component, includes two *SELECT* statements. The first *SELECT* statement doesn't include any conditions and obtains object from *Class 1* in object variable *t*. The inner *SELECT* statements includes *anti-join* conditions. Before entering the second loop, there is a variable initialised to *False*.

This variable will become true if the *SELECT* statement finds any object from *Class 2* which satisfies the *anti-join* condition. The object variable is t an object variable from *Class 1*.

There are a number of different ways to implement *anti-join* applications. This thesis considers *anti-join* applications which can be implemented with *not-exist*. Note that any *not found* statement can be re-written into an equivalent *not-exist* statement. For example, *if it is not-found then EXIT the loop* is equivalent to *if it is not-exist then EXIT the loop*.

Algorithm 9: Input component 1
Anti-Join by Variable

```
1 for each t in (SELECT * FROM Class 1) do
2     Found = False
3     for each s in (SELECT * FROM Class 2) do
4         if γ' [< s.s_1, t.t_1 >, ..., < s.s_n, t.t_n >] then
5             Found = True
6             Exit the loop
7     if not Found then
8         P r o c e s s i n g(t)
```

Algorithm 10 is presented, to show the input component of this rule.

Algorithm 10: Example 3/Non-Optimise version

```
1 for each t in (SELECT * FROM Department) do
2     Found = False
3     for each s in (SELECT * FROM Employee) do
4         if t.Department_id = s. Department_id then
5             Found=True
6             Exit the loop
7     if not Found then
8         Write (t)
```

The output of Algorithm 10 is all of the department-id from the *Department* class which are not related to any department-id in the *Employee* class. In other words, the application will find all the department-id from the *Department* class which are not the same as the department-id in *Employee* class. This is an example of anti-join application.

Output Component

The output component for *anti-join* is implemented by one *SELECT* statement using our approach. The *left-outer-join* is used in the output component of this rule to select the objects which satisfies the *anti-join* conditions and makes it unnecessary to transfer them to the client side. The following algorithm can be used as an output component for two different implementations of the *anti-join*'s input components.

The *left outer join* condition consists of the object variables t and s which are the object variables from *Class1* and *Class2* respectively.

Algorithm 11: Output component

Anti-Join by Left outer join

```
1 for each p in (SELECT * FROM Class1 Left Outer Join Class2 on
2 γ' [< t1, s1 >, < t1, s2 >, ..., < tn, sn >]) do
3     if p is Null then
4         Processing(p)
```

Implementing the *anti-join* algorithm is based on the input component structure by the *nested loop join* as a possible case. However, for large databases it is not efficient to use this structure because navigating through associations by nested loops, is based on comparing each object with an entire class. The output component executes faster than the input component. The output component operates on object variable p. The following algorithm is an example of the output component of this rule.

Algorithm 12: Example 3/Optimised version

```
1 for each p in (SELECT * FROM Department left outer join Employee on
2 Department.Department_id = Employee.Department_id) WHERE
  p.Employee.Department_id is Null do
3     Write p
```

Input Component 2

There are different ways to implement *anti-join* applications. However, there are two common implementations mostly presented by the application developers. The first style of anti-join application is described in Section 3.3.2. The second common possible input component for the *anti-join* is presented here. The application programmer can design an *anti-join* algorithm by using a *counter* in the body of the application. In this case, there are two *SELECT* statements. Inside the first *SELECT* statement there is not any condition and the objects from this class will pass to the object variable *t*. The inner *SELECT* statement includes the *anti-join* condition and the *counter*. The counter is checked to find the object from the first class which is the same in the other class. In this case the counter would not be zero at the end of the loop, so the objects which are not the same, are the output.
The output of Algorithm 13 is the same as the output for Algorithm 9 which is Algorithm 11.

The variable s_i is a member of *Class2*. i is a positive number so the variable s_i is one of the objects from *Class2*. An example of *Class2*.s_1,$t.t_1$: Employee.employee_id= t.employee_id.

Algorithm 13: Input component 2

Anti-Join by Counter

```
1 Count=0
2 for each t in (SELECT * FROM Class1) do
3     Count= (SELECT Count(*) FROM Class2
4     WHERE
5     γ' [< Class2.s_1, t.t_1 >, ..., < Class2.s_n, t.t_n >])
6     if Count=0 then
7         P r o c e s s i n g(t)
```

The problem of different implementations of the input component is already solved by software patterns which will be presented in Chapter 4.

3.4 Aggregation

For aggregation applications, one input and one output components is proposed. These input and output component are applicable to different aggregation functions such as Min, Max, Average and Count.

3.4.1 Input Component

The most common implementation of aggregation queries is modelled in Algorithm 14. Algorithm 14 is an input component of aggregation applications. In this implementation of aggregation applications, developers are using nested SELECT statements to find the appropriate objects for aggregation function from a class. Two nested SELECT statements are presented by OO programmers to build the aggregation functions, but this is not suitable for the implementation of OO applications. This nested loops implementation, takes an object variable from the outer loop and the value of the property of this object is transferred to the condition of the other loop. In this structure, an alias for the class is used to run the aggregation function for representative objects from *Class*. The aggregation rule is based on finding desired objects in a class and then applying the aggregation function to them. F(x) is a *COUNT(*)* in the input component and *COUNT(Member$_i$)* in the output component, for counting similar objects from a class of objects. This structure for aggregation

queries transfers all the objects from the server side to the client. Then the *Class* is considered as a group of objects.

In line 2, the RESULT is a variable which stores results of second loop. It is assumed that F(x) is an aggregation function such as Min, Max, Average and Count. In this function object variable s is used. It is assumed that WHERE condition is a condition between members of Class1 and Class2. For instance it can be Class1.Member$_i$=Class2.Member$_i$.

Algorithm 14 is an input component of aggregation queries.

Algorithm 14: Input component
Aggregation with nested loop

```
1 for each t in (SELECT Member_i FROM Class 1) do
2     for each s in (SELECT F(x) FROM Class 2 as RESULT
3     WHERE
4     t.Member_i= Class 2.Member_j) do
5         Processing < t, s >
```

An example of the input component of this rule is presented as Algorithm 15.

Algorithm 15: Example 4/Non-Optimised version

```
1 for each t in (SELECT * FROM Department) do
2     for each s in (SELECT count(*) as TOTAL FROM Employee
3     WHERE
4     t.Department_id = Employee. Department_id) do
5     Write < t, s >
```

Algorithm 15 consists of two nested loops. The outer loop selects all objects from

class *Department* and passes the result to the second loop. The inner select statement is responsible to find all the objects which satisfy the condition (t.Department_id = Employee. Department_id). The output of this algorithm is the result of the *count* function which counts the number of the same department-id from both the *Department* and *Employee*.

3.4.2 Output Component

The *Group by* clause is used to group the necessary objects and transfer them to the client side, in the output component of this rule. Therefore, compared to the input component, less objects will be transferred from the server side. F(x) can be replaced by any of the aggregation functions. For instance, *Count(*)* can be used instead of F(x) in the output component to count objects.

Algorithm 16: Output component
Counting and grouping objects

```
1 for each p in (SELECT Member_i, F(x) as RESULT FROM Class 1, Class 2
2 WHERE
3 γ' [< s_1, t_1 >, ..., < s_n, t_n >]) Group by Member_i FROM Class 1 do
4 |  Processing < p >
```

Algorithm 16 minimises the run-time of an aggregation application. The reason is using of *Group by* clause in the output component. The *Group by* clause is used to divide the objects into groups. In the input component without *Group by*, the entire *class* is considered as one group. An example of the output component is presented in Algorithm 17.

Algorithm 17: Example 4/Optimised version

```
1 for each p in (SELECT count(Department.Department_id) as TOTAL
2 FROM Department, Employee
3 WHERE
4 Department.Department_id = Employee. Department_id
5 GROUP BY
6 Department.Department_id do
7 |  Write (p)
```

All of the objects with the same department *_id* from both the *Department* and the *Employee* classes are selected. The output of this application is the result of counting.

Chapter 4

Correctness of Transformation Rules

4.1 Introduction

The Hoare rules or the FloydHoare logic is a formal method to prove the correctness of computer programs. This method is based on logical rules and it has been used by computer scientists and logicians since 1969 [AA78].

The *Floyd Hoare logic* is used to prove the correctness of each rule. Also this method shows that in each rule both input and output components achieve the same output. Writing the same invariant for the input and output component of each rule allows us to prove the correctness of each rule. The algorithms under analysis have been rewritten using *While* in order to be consistent with the *Floyd Hoare logic* format which is presented in [AA78]. Based on the definition of the *Floyd Hoare logic*, the invariant of each algorithm must represent the structure of the algorithm. For instance, the invariant for join algorithm must represent the join operation based on the structure of that rule. An invariant is invented for both input and output components of each rule. This invariant must be true before and after the *while* loop. This means that before and after entering the *While* loop, the structure of the algorithm is the same. Therefore, the whole algorithm is correct. If the input and output components have the same invariant, then the both algorithm representing the same operation. The correctness of each individual algorithm is proved by writing an invariant for each algorithm. The equality of both components is proved by verifying the same invariant for each pair of the rules.

In this section, first the Floyd Hoare logic is explained briefly, then the proof for the first group of the applications (iterating over a class of objects) along with the method of verification of the rule is described. In the following algorithm, everything

inside the curly brackets is related to loop invariants and pre-condition of the Floyd Hoare logic and must not be considered as a part of the algorithm.

Floyd Hoare logic or Hoare rule is as follow:

$$\frac{\{P \wedge B\}S\{P\}}{\{P\}while(B)do(S)\{\neg B \wedge P\}}$$

Where P is the loop invariant which should to be preserved by the loop body S. B is the condition of the application. After the loop is finished, this invariant P still holds, and moreover ¬ B must have caused the loop to end. Loop invariants are used to prove the correctness of the loop properties. Assertions on state enters the loop and guarantees to be true at every iteration of the loop. The invariant is the post-condition for the loop on exit. The logic includes two separate parts (some logics are placed over the line and some others underline). The correctness of the under the line logics is automatically proven by proving the correctness of the over the line logics. As a result all of the functions are correct. *Hoare logic* is complete, so it is a reliable method for proving the algorithms.

$$\frac{\{Loop\ Invariant \wedge Condition\}Body\{Loop\ Invariant\}}{\{Loop\ Invariant\}\ while(Condition)\ do\ (Body)\{\neg Condition \wedge LoopInvariant\}}$$

The *Hoare logic* is applied to all of the proposed algorithms as follow:

4.2 Iteration over one class of object

Algorithm 18, is an input component of the first rule which has been rewritten with *pseudo-code* using *while* and includes the invariant. The correctness of this input and its output component is described to show the method of proving the other rules. In line 2, all t objects are selected from C. In line 3, before entering the *while* loop, there is a condition and an invariant that are true at that stage of the algorithm. The condition 't <> Nil' proves if there is at least one object in class 1 before entering the *while* loop or not. This condition is true before entering the *while* loop. If the

condition becomes false then there is no object in *Class*, and the loop will be skipped. The invariant before and after the *while* loop is the same: '$\forall t \in R : \varphi\,[t]$'. This means only objects that satisfy the condition of $\varphi\,[t]$ will go into the output. This invariant is true before entering the *while* loop and also after exiting or skipping the *while* loop. The object which satisfies condition $\varphi\,[t]$, will be added to the result set R.

It is assumed that R is the result set

It is assumed that C is SELECT * FROM class 1

It is assumed that t is a set of objects from class 1

It is assumed that $\varphi\ [t.t1, t.t2, ..., t.tn] = \varphi\ [t]$

Algorithm 18: Input component
Iterating over a class of object

```
1 R:= Ø
2 Get (t, C)
3 {(t <> Nil) ∧∀t ∈ R :φ [t]}
4 while t <> Nil do
5 |   if φ [t] then
6 |   |   R:= R∪ [t]
7 |   Get (t, C)
8 {∀t ∈ R :φ [t]}
```

Based on the *Floyd Hoare logic*, the logic which is placed above the line for the input component is as follows:

{Condition} ∧ {Invariant} [*Body*] {Invariant}

{(t <> Nil) ∧∀$t \in R : \varphi\,[t]$ }[If $\varphi\ [t]$ Then R:= R∪ [t] , Get (t, C)]
{$\forall t \in R : \varphi\,[t]$}

This is always true as explained in Section 4.1. Therefore the second part is automatically true:

$\{Invariant\} while(Condition) body \{\neg Condition \wedge Invariant\}$

$\{\forall t \in R : \varphi\,[t]\}$ while (t <> Nil) [If φ $[t]$ Then R:= R$\cup$ $[t]$, Get (t, C)]
{ (t : Nil) $\wedge \forall t \in R : \varphi\,[t]$ }

The correctness of the input component is proved because the same invariant is achieved after finishing the loop. The next stage is to prove that the output is correct by applying the *Floyd Hoare logic* to the output. Also, the same invariant from both patterns and their outputs is achieved to make the assumption that each pattern and its rule are doing the same thing. Algorithm 19 is the output component for iterating over a class of object, which has been rewritten with a *While* loop.

It is assumed that C' is SELECT * FROM class 1 WHERE φ $[t1, t2, ..., tn]$

Algorithm 19: Output component
Rule for iteration over a class of object

```
1 R:= 0
2 Get (t, C')
3 {(t <> Nil) ∧∀t∈ C' : φ[t] ∧∀t ∈ R :φ[t] }
4 while t <> Nil do
5 |   R:= R∪t
6 |   Get (t, C')
7 {∀t ∈ R :φ[t]}
```

The *Floyd Hoare logic* is applied to the output component as follow.

{Condition} $\wedge$ {Invariant} [*Body*] {Invariant}

{(t <> Nil) $\wedge \forall t \in C' : \varphi\,[t] \wedge \forall t \in R : \varphi\,[t]$ } [R:= R$\cup t, Get(t, C')$]
$\{\forall t \in C' : \varphi\,[t] \wedge \forall t \in R : \varphi\,[t]$ }

This is always true as explained in Section 4.1. Based on the Floyd Hoare Logic, the second part is automatically true because:

$\{Invariant\} while(Condition) body\{\neg Condition \wedge Invariant\}$

$\{\forall t \in C' : \varphi[t] \wedge \forall t \in R : \varphi[t]\}$ while ((t <> Nil) $\wedge \forall t \in C' : \varphi[t]$) [R:= R$\cup t, Get(t, C')$] $\{$(t:Nil) $\wedge \forall t \in C' : \varphi[t]\}$

This is always true. Before entering the *while* loop, both condition and the invariant are true because there must be at least one object to enter the *while* loop. Each object t can satisfy the condition of $\varphi[t]$. After exiting or skipping the loop, all the objects in set R satisfy the condition of $\varphi[t]$. Therefore, based on the *Floyd Hoare logic*, this rule is proved. In addition, both the input and output components of the rule include the same invariant, so this can prove the equivalence of both algorithms. The same method is used to show the correctness of the other patterns. The term $\forall t \in R : \varphi[t]$ is the invariant for both the input and the output component of the first rule. Therefore, both algorithms perform the same operation and will achieve the same output.

4.3 Traversal of association

4.3.1 Join Traversal

The same method is used to prove the correctness of the other rules. The input and the output components of the *join traversal* has been rewritten by a *while* loop to be matched with the Floyd Hoare logic. The invariant introduced, is based on the body of the code. Algorithm 20 is based on the input component for iteration over two classes of objects.

In the first line of Algorithm 20, R is the result set which is empty. In line 2 all the objects from *Class1* and *Class2* are selected. The variable t is a set of objects from class1 and s and is a set of objects from *Class2*. Line 3 presents the invariant of this algorithm. This invariant declares that, for all objects r in the result set R the following statement is correct:

Each object t in *Class1* which satisfies the condition $\varphi[t]$ concatenate with each object s in *Class2* which satisfies condition ($\gamma[s]$ + $\gamma'[s]$). + does the AND

operation. So that the objects which satisfy the join condition from *Class1 Class2*, concatenates together and are added to the result set R. Lines 4 to 10 are the *while* loop.

This loop presents the input component of the join traversal which is already explained in Section 3.3.1. Before entering the *while* loop, the invariant is correct. After entering or skipping the loop, the invariant which is presented in line 11 is still correct. This is because the algorithm only accepts the objects that satisfy the *join* condition.

The definition of the following assumptions has already been given in 3.3.1.

It is assumed that r= t∧s

It is assumed that C1(Class1) is SELECT * FROM Class1 WHERE $\varphi\ [t]$

It is assumed that C2(Class2) is SELECT * FROM Class2 WHERE $\gamma\ [s]$ AND $\gamma\prime\ [s]$

It is assumed that $\varphi\,[t] = \varphi\ [t_i] : \varphi\ [t_1, t_2, ..., t_n]$

It is assumed that $\gamma\ [s] = \gamma\ [s_i] : \gamma\ [s_1, s_2, ..., s_n]$

It is assumed that $\gamma\prime\ [s] = \gamma\prime\ [s_i.t_j]$: $\gamma'\ [< s_1.t_1 >, < s_2.t_1 >, ..., < s_n.t_m >]$

Algorithm 20: Input component
Iterating over two classes of object

```
1  R:= ∅
2  Get (t, C1) , Get (s, C2)
3  {(t <> Nil)} ∧ ∀r ∈ R : ∃t ∈ C1| φ[t] ∧ ∃s ∈ C2| ( γ [s] + γ′ [s] ) ∧ r= t‖s )
4  while t <> Nil do
5      if φ [t] then
6          while s <> Nil do
7              if γ [s] + γ′ [s] then
8                  R:= R∪(t ∧ s)
9              Get (s, C2)
10     Get (t, C1)
11 { ∀r ∈ R : ∃t ∈ C1| φ[t] ∧ ∃s ∈ C2| ( γ [s] + γ′ [s] ) ∧ r= t‖s }
```

Applying the *Floyd Hoare logic* to the input component as follow.

{Condition} $\wedge$ {Invariant} [*Body*] {Invariant}

{(t <> Nil) $\wedge \forall r \in R : \exists t \in C1|\ \varphi\,[t] \wedge \exists s \in C2|$ (γ [s] + $\gamma\prime$ [s]) $\wedge$ r= t$\|$s [If γ [s] + $\gamma\prime$ [s] then R:= R$\cup$(t $\|$s), Get ($s, C2$), Get ($t, C1$)] $\forall r \in R : \exists t \in C1|\ \varphi\,[t] \wedge$ $\exists s \in C2|$ (γ [s] + $\gamma\prime$ [s]) $\wedge$ r= t$\|$s

This is always true according to Section 4.1. The second part of the logic is applied and it is automatically true because:

$\{Invariant\}while(Condition)body\{\neg Condition \wedge Invariant\}$

$\forall r \in R : \exists t \in C1|\ \varphi\,[t] \wedge \exists s \in C2|$ (γ [s] + $\gamma\prime$ [s]) $\wedge$ r= t$\|$s while {(t <> Nil) [If γ [s] + $\gamma\prime$ [s] then R:= R$\cup$(t $\|$s), Get ($s, C2$), Get ($t, C1$)] {(t: Nil) $\wedge \forall r \in R : \exists t \in C1|$ $\varphi\,[t] \wedge \exists s \in C2|$ (γ [s] + $\gamma\prime$ [s]) $\wedge$ r= t$\|$s

The algorithm has been rewritten with the *while* loop to prove the correctness of the output component of the second rule. In algorithm 21, R is the result set. In line two, the object set from joining *Class1* and *Class2* are selected.

In line three, the first precondition is placed. The precondition of this algorithm is r not null and variable x and comes from joining *Class1* and *Class2* and satisfies the *join* condition. The rest of the line is the invariant of this algorithm. The invariant shows that all objects in the result set satisfy the *join* condition. Line three is true before entering the *while* loop. Lines four to six are the body of the output component which is explained earlier in section 3.3.1. In the case of the exit or skip loop, line seven which includes the invariant should be true. This line shows that all objects in the result set R satisfy the *join* condition. If this is always true, then the whole algorithm is true.

It is assumed that (C1 $\bowtie$ $C2$) =SELECT * FROM Class1 **Join** Class2 on γ' [t] **Where** φ [t] $\|\gamma$ [s]

It is assumed that γ' [t] is γ' [$< t_1, s_1 >, < t_1, s_2 >, ..., < t_n, s_n >$]

It is assumed that $\varphi\ [t_1, ..., t_n]$

It is assumed that $\gamma\ [s_1, s_2, ..., s_n]$

It is assumed that Φ(r) is γ' $[t] \wedge \varphi\ [t]\ \|\gamma\ [s]$

It is assumed that pre-Condition is $\forall x \in (\text{C1} \bowtie C2) : \Phi(x)$

Algorithm 21: Output component
Iterating over two classes of object

```
1 R:= Ø
2 Get (r, C1 ⋈ C2)
3 { ∀r <> Nil∧ ∀x ∈ (C1 ⋈ C2) : Φ(x) ∧ ∀r ∈ (C1 ⋈ C2) : Φ(r) }
4 while r <> Nil do
5 |   R:= R ∪ [r]
6 |   Get (r, C1 ⋈ C2)
7 { ∀r ∈ (C1 ⋈ C2) : Φ(r) }
```

The *Floyd Hoare logic* is applied to the output component as follow.

{Condition} $\wedge$ {Invariant} [*Body*] {Invariant}

{ $\forall r <> Nil \wedge\ \forall x \in (\text{C1} \bowtie C2) : \Phi(x)$} $\wedge$ {$\forall r \in (\text{C1} \bowtie C2) : \Phi(\text{r})$} [R:= R $\cup$ r, Get $(r, C1 \bowtie C2)$] {$\forall r \in (\text{C1} \bowtie C2) : \Phi(\text{r})$}

The second part of the logic is automatically true as follow:

$\{Invariant\} while (Condition) body \{\neg Condition \wedge Invariant\}$

{$\forall r \in (\text{C1} \bowtie C2) : \Phi(\text{r})$} while ($\forall r <> Nil \wedge\ \forall x \in (\text{C1} \bowtie C2) : \Phi(x)$) [R:= R $\cup$ r, Get $(r, C1 \bowtie C2)$] {$(\forall r : Nil \wedge\ \forall x \in (\text{C1} \bowtie C2)\ \neg\ \in \Phi(x)) \wedge \forall r \in (\text{C1} \bowtie C2) : \Phi(\text{r})$}

The invariant of the input component is $\forall r \in R : \exists t \in C1|\ \varphi\,[t] \wedge \exists s \in C2|$ ($\gamma\ [s]\ +\ \gamma\prime\ [s]$) $\wedge$ r= t$\|$s performs the same operation as the invariant of the output component: $\forall r \in (\text{C1} \bowtie C2) : \Phi(\text{r})$. Both invariants are explaining the *join* operation. The input component is implemented with the nested loop and the output component is implemented with the *join* operation. Therefore both input

and output components are performing the same operation.

4.3.2 Anti-join Traversal

Algorithm 22, presents the *anti-join* input component by the *while* loop with the invariant. Line one, presents the result set R. Objects from *Class1* are selected in line two. Line three presents the precondition and the invariant. There is at least one object in *Class1* for precondition to be true. The invariant is true if object-set t does not satisfy the *anti-join* condition ψ $[s.t]$. Lines 4 to 11, present the body of the *anti-join* input component which is discussed in Section 3.3.2. Line 3 is correct before entering the *while* loop as the precondition is true. At the end of the algorithm the invariant is still true. This means the whole algorithm is correct. That is because after exiting the loop, the invariant still holds and moreover if the precondition in line 3 does not hold, the loop would skip. This is proved by applying the Floyd Hoare logic formal system to the algorithm.

Algorithm 22: Input component
Antijoin

```
1  R:= Ø
2  Get (t, C1)
3  Get (s, C2)
4  { t <> Nil ∧ ∀t∈ R | t ∉ ψ [s.t] }
5  while t <> Nil do
6      Found= False
7      Get (s, C2)
8      while s <> Nil do
9          if ψ [s, t] then
10             Found = True
11         Get (s, C2)
12     if Not Found then
13         R:= R ∪[t]
14     Get (t, C1)
15 { ∀t∈ R | t ∉ ψ [s.t] }
```

The *Floyd Hoare logic* is applied to the input component as follow.

{Condition} $\wedge$ {Invariant} [*Body*] {Invariant}

{t <> Nil} $\wedge$ {$\forall t \in$ R | t $\notin \psi$ [$s.t$] } [Found = True, Get $(s, C2)$, R:=R $\cup(t), Get(t, C1)$] {$\forall t \in$ R | t $\notin \psi$ [$s.t$]}

The second part of the logic is applied as:

$\{Invariant\} while(Condition) body \{\neg Condition \wedge Invariant\}$

{$\forall t \in$ R | t $\notin \psi$ [$s.t$]} while ({t <> Nil}) [Found = True, Get $(s, C2)$, R:=R $\cup(t), Get(t, C1)$] {t: Nil $\wedge \forall t \in$ R | t $\notin \psi$ [$s.t$]}

The output component of the *anti-join* has been rewritten with *while* and the invariant which is presented by the following algorithm. In this algorithm, result-set R is empty before starting the loop. Object-set r is selected from the left-outer-join operation between Class1 and Class2($(C1 ⟕ C2)$). Line three includes a precondition and an invariant of the algorithm. For the precondition to be true, there should be at least one object r. If this condition becomes false the the loop is skipping and the invariant in line seven should hold. If the precondition becomes true then the invariant in line three must held. The invariant is true because after the precondition is held, each object in $C1 ⟕ C2$, must satisfy the *anti-join* condition which is $\omega\,[r]$. Therefore before and after the *while* loop the invariants are held so the whole algorithm is true. The logic is applied to the algorithm to prove the correctness.

It is assumed that $(C1 ⟕ C2)$ is SELECT * FROM Class1 **LEFT OUTER join** Class2 on $\omega\,[r]$.

It is assumed that $\omega\,[r]$ is anti-join condition $Class2.Memberj = Class1.Memberi$.

The way that the anti-join symbol is used in Algorithm 23: "$(C1 ⟕ C2)$", is to show that the join is left-outer-join and not rigth. That's why the join symbol is placed near C1 not C2.

Algorithm 23: Output component
Antijoin

```
1 R:= Ø
2 Get ( r, (C1⋈ C2))
3 { r <> Nil ∧ ∀r ∈ (C1 ⋈ C2) :ω [r] }
4 while r <> Nil do
5 |   Get ( r, (C1⋈ C2))
6 |   R:=R ∪r
7 while R <> nil do
8 |   Get (r, C1)
9 { ∀r ∈ (C1⋈ C2) :ω [r] }
```

The *Floyd Hoare logic* is applied to the output component as follow:

{Condition} ∧ {Invariant} [*Body*] {Invariant}

{r <> Nil} ∧ {$\forall r \in (C1 \bowtie C2) :\omega [r]$} [Get (r, (C1⋈ $C2)), R := R \cup r$]{$\forall r \in (C1 \bowtie C2) :\omega [r]$}

The second part of the logic is:

$\{Invariant\} while (Condition) body \{\neg Condition \wedge Invariant\}$

{$\forall r \in (C1 \bowtie C2) :\omega [r]$} while {r <> Nil} [Get (r, (C1⋈ $C2)), R := R \cup r$]{$r : Nil \wedge \forall r \in (C1 \bowtie C2) :\omega [r]$ }

The invariant of the input component is $\forall t \in$ R | t $\notin \psi$ $[s.t]$ which is exactly the same as the invariant in the output. In the output component, the anti-join operation is described by the *left-outer-join* and it is $\forall t \in$ R | t $\notin \psi$ $[s.t]$. Both these invariants are an explanation of the *anti-join*. So, by achieving the same invariants, the similarity of the input and output components are proved.

4.4 Aggregation

Algorithm 24 presents the input component of an aggregation query which is already presented in Section 3.4.1 by a *While* loop. This algorithm is using the *While* loop. R is the result set. Precondition and invariant is presented in line three. Lines four to seven are the body of the aggregation application. If the precondition is true then the invariant must be true before iterating the while loop. The invariant is true if all the objects in the result set satisfy the aggregation conditions. After the loop is finished, the invariant in line eight is held. This algorithm can be used for any aggregation function: Min, Max, Average and Sum.

It is assumed that C is (SELECT Member$_i$ FROM Class 1)
It is assumed that C' is (SELECT $F(y)$ FROM Class 2 WHERE condition)
It is assumed that t is an object variable from C
It is assumed that s is an object variable from C'
It is assumed that $F(y)$ is an aggregation function, for example: Count(*)
$\forall r \in R : \exists s \in C' \mid F(y)$: for all r in R there is "s" exists in C'

Algorithm 24: Intput component
Aggregation

```
1 R:= ∅
2 Get (t, C) ,
3 {(t <> Nil)} ∧ (s <> Nil)} ∧ ∀r ∈ R : ∃s ∈ C'| F(y)
4 while t <> Nil do
5 |   while s <> Nil do
6 |   |   Get (s, C')
7 |   |   R:= R∪s
8 |   Get (s, C')
9 { ∀r ∈ R : ∃s ∈ C'| F(y) }
```

Applying the *Floyd Hoare logic* to the input component as follow:

{Condition} ∧ {Invariant} [*Body*] {Invariant}

{ {(t <> Nil)} ∧ (s <> Nil)} ∧{$\forall r \in R : \exists s \in C' |\ F(y)$ } [R:= R∪ s, Get (s, C')] { $\forall r \in R : \exists s \in C' |\ F(y)$ }

Applying the second part of the logic:

$\{Invariant\} while(Condition) body \{\neg Condition \wedge Invariant\}$

{ $\forall r \in R : \exists s \in C' |\ F(y)$ } while ({ {(t <> Nil)} ∧ (s <> Nil)}) [R:= R∪ s, Get (s, C')] { (t:Nil) ∧$(s : Nil)$}

The output component of the aggregation query has been rewritten by the *while* loop and it includes the invariant. The output component uses *group by* clause. The invariant which is presented in line three is considered all the objects in C' which satisfies the aggregation condition. The same invariant is held after finishing the *while* loop. If the precondition becomes true, it means that there is at least an object in *C'*, then the invariant must be true. After finishing or skipping the loop, the invariant is still held which means the whole algorithm is correct. The *while* loop is checked if the objects follow the aggregation function or not. If yes, the objects would add to the result set.

It is assumed that *C"* is SELECT Member_i, F(y) FROM Class1, Class2 WHERE condition GROUP BY Member_i

It is assumed that F(y) is an aggregation function, for example: Count(*)

It is assumed that p is an object variable from *C"*

Algorithm 25: Output component
Aggregation

```
1 R:= ∅
2 Get (t,C")
3 { (p <> Nil) ∧∀r ∈ R : ∃p ∈ C" | F(y) }
4 while (t <> Nil) do
5 |  R:= R∪p
6 Get (p, C")
7 { ∀r ∈ R : ∃p ∈ C" | F(y) }
```

The *Floyd Hoare logic* is applied to the output component as follow:

{Condition} $\wedge$ {Invariant} [*Body*] {Invariant}

{ (p <> Nil)} $\wedge$ $\{\forall r \in R : \exists p \in C" |\ F(y)\ \}$ [R:= R$\cup p, Get(p, C")$] { $\forall t \in C"$: $F(y)$ }

Applying the second part of the logic:

$\{Invariant\} while(Condition) body \{\not Condition \wedge Invariant\}$

{ $\forall r \in R : \exists p \in C" |\ F(y)\ \}$ while ({ (p <> Nil)}) [R:= R$\cup p, Get(p, C")$] {(p: Nil) $\wedge$ $\forall r \in R : \exists p \in C" |\ F(y)\ \}$

All of the algorithms are overwritten using *while* loop to be able to use the *Floyd Hoare logic*. The invariant of both the input and the output components of the rule are the same. It can be concluded that these two algorithm are the same by having the same invariant for the input and output component of this rule. { $\forall r \in R : \exists s \in C' |\ F(y)\ \}$ is the invariant of the input component and the same invariant with different SELECT statements is used for the output component: { $\forall r \in R : \exists p \in C" |\ F(y)\ \}$. As explained earlier, F (y) is referring to the aggregation function. This means that both invariants of the input and the output algorithm are performing the same functions and as a result both invariants and both algorithms are same.

Chapter 5

Code Templates

5.1 Introduction

An input component of any transformation rule is a non-optimised version of the OR application. This application is written by an OO developer and is based on iteration over a class of objects. An input component is required to use the transformation rules. The problem is that the application developers can design applications in different ways. Different rules are verified to cover all possible cases.

To solve this problem, certain Java patterns were proposed which individual JPA programmers can use to write an OO application. By having certain patterns for the input components of the rules, the analysis of the source code will be easier. The proposed templates can be used to match the implementation of the current OO programs. The majority of OR database applications are combinations of the following styles, so the transformation rules are applicable to them.

Each transformation rule presented in Chapter 3 includes an input and output component. The input component which is a non-optimised version of the application has to find the relevant transformation rule. In this section, each input component is presented by a code template. This makes the process of finding the correct output component easier.

The following styles are based on the Java programming language and are similar to most of the other OO languages. Application programmers can match their applications with the following Java templates. Depending on the application, the name of the classes of objects, the aggregation functions, the relational conditions and the non-relational conditions will change. OO programmers have to replace the

statement inside '< >', with the appropriate statement of their code. The statements in '[ ]' are optional. Other parts of the template remain unchanged.
The syntax of relational and non-relational conditions are already explained in Chapter 3, to avoid repetition they will not be explained here again.

5.2 Filtering Template (F.Temp)

The input component of the first transformation rule is consistent with the following template.

```
{
  Query query1 = em.createQuery(
     <Any SQL SELECT statement>);
     List list1 = query1.getResultList();
     Iterator iterator1= list1.iterator();
      while(iterator1.hasNext()){
       {
          if <CONDITIONS> then
             <Java code>;
       }
}
```

Where <CONDITIONS> is a type of filtering condition for the class which filter the output. For instance: *l.partsupp*= 900. This condition means that, the objects in *l.partsupp* which have equal value to "900" will be selected for the output result from the class.

5.3 Traversal of Association

5.3.1 Join Traversal of Association Template (TA.Temp)

The output component which is presented in Section 3.3.2 is the optimised version of the application, only if the input component matches the following style.

```
{
  Query query1 = em.createQuery(
      <SQL SELECT statement from CLASS1> +[ NON-RELATIONAL CONDITIONS of CLASS1]);
      <GET VARIABLE> ;
      List list1 = query1.getResultList();
      Iterator iterator1= list1.iterator();
      while(iterator1.hasNext())
      {
        <VARIABLE> =  list1.getInt(1);
         if <CONDITIONS> then
         {
        Query query2 = em.createQuery(
          <SQL SELECT statement from CLASS2>
             <RELATIONAL CONDITIONS>);
        List list2 = query2.getResultList();
        Iterator iterator2= list2.iterator();
        while (iterator2.hasNext())
          {
                if <RELATIONAL CONDITIONS of CLASS2> then
                   <Java code>;
          }
        }
      }
}
```

Where <RELATIONAL CONDITIONS> are *join* conditions. An example of <RELATIONAL CONDITIONS> is: l.l_suppkey = s.s_suppkey. This *join* condition find the values of l_suppkey from *Lineitem* class which are same as s.s_suppkey in *Supplier* class. The $< VARIABLE >$ is used to retrieve the results in <*Java code*>.

An example of <RELATIONAL CONDITIONS> is: $l.l\_suppkey = s.s\_suppkey$. It is assumed that l is an object variable from class1 and s is an object variable from class2.

An example of <NON-RELATIONAL CONDITIONS> : $l.l\_suppkey = 423$ or $s.s\_suppkey = 533$

5.3.2 Anti-Join Traversal of Association Template (AJ.Temp)

If the input component of the rule matches any of these two *anti-join* styles then it can be modified according to the rule presented in Section 3.4.3.

Style 1: This style of *anti-join* input component uses a variable. In the following pattern, VAR is referred to as a variable. The variable is *false* before entering the second *while* loop. The variable will become *true* for each object which satisfies the <*Anti-join Condition*>. As a result, only objects which do not satisfy the <RELATIONAL CONDITIONS> would be in the output.

```
{
  Query query1 = em.createQuery(
      <SQL SELECT statement from CLASS1>);
      GET VAR = false ;
      List list1 = query1.getResultList();
      Iterator iterator1= list1.iterator();
      while(iterator1.hasNext()){
        Query query2 = em.createQuery(
        <SQL SELECT statement from CLASS2>
          where
           <ANTI-JOIN CONDITION>;
       List list2 = query2.getResultList();
       Iterator iterator2= list2.iterator();
       while (iterator2.hasNext())
        {
        if <ANTI-JOIN CONDITION>{
          VAR = True;
          Exit;
        } }
       if VAR==false
             {
               <Java code>;
             }
        }
}
```

Style 1 can also be defined as follows:

```
{
  Query query1 = em.createQuery(
      <SQL SELECT statement from CLASS1>);
      List list1 = query1.getResultList();
      Iterator iterator1= list1.iterator();
      while(iterator1.hasNext())
      { Query query2 = em.createQuery(
        <SQL SELECT statement from CLASS2>
          where
           <Anti-join Condition>;
       List list2 = query2.getResultList();
       Iterator iterator2= list2.iterator();
       if List2.IsEmpty()
             {
               <Java code>;
             }
        }
}
```

Style 2: In this style, the programmer is using a counter. The counter checks if any object satisfies the anti-join condition.

```
{
    Count=0
    Query query1 = em.createQuery(
      <SQL SELECT statement from CLASS1>);
      List list1 = query1.getResultList();
      Iterator iterator1= list1.iterator();
      while(iterator1.hasNext())
    {
    Query query2 = em.createQuery(
      SELECT Count(*) FROM <CLASS2>
       where
        <Anti-join Condition>;
        int Count = query1.getSingleResult()
        if Count == 0
```

```
     {
       <Java code>;
     }
}
}
```

5.4 Aggregation Template (AG.Temp)

The template used for the aggregation rule is presented below. If the input component matches the following template, the aggregation output component which is presented in 3.5.2 is the optimised version of application. F(x) can be any type of the aggregation function: Min, Max, Sum, Avg or Count.

[CONDITION] is not compulsory for being in the template. It depends on the application. [CONDITION] is any condition between two classes. For example: l.emp-num = s.emp-num.

```
{ Query query1 = em.createQuery
 ("SELECT <Memberi> FROM <CLASS 1>");
   List list1 = query1.getResultList();
   Iterator iterator1= list1.iterator();
   while(iterator1.hasNext())
    {
    Query query2 = em.createQuery
    ("SELECT <F(x)>
     FROM <CLASS 2>
      WHERE
      [CONTITION]");
        List list2 = query2.getResultList();
        Iterator iterator2= list2.iterator();
         while (iterator2.hasNext())
        {
          <Java code>;
        }
     }
}
```

5.5 *n* Associations Template

The application developers distinguishes the template of their application in order to use one of the rules explained in Chapter 3. The n Associations Template rule are described in this section for the developers to build the application template step by step. Then the rules are applied to the application and as a result the optimised version of the application is obtained.

It is assumed that F is : Filtering, C is : Condition, J is : Java Code, JC is : Join Conditions, V is : Variable, A is : Array, AGC is : Aggregation Conditions and OV is an Object Variable. The OV, keeps the results from one template and passes it to the other template. The object variable takes the results from each template and passes it to the next template. At the end of each template, the object variable is updated to the new object variable which includes new results from the current template and this object template is ready to be used in the next template. So generally all of the templates can be presented as *short templates* as follows:
$\mathrm{T}_F < F, C, OV, J >$
$\mathrm{T}_{TA} < F_1, F_2, C, JC, OV, J >$
$\mathrm{T}_{AJ} < F_1, F_2, C, OV, J >$
$\mathrm{T}_{AG} < A, F, AGC, OV, J >$
The template of each association has to be created and linked together by an object variable for making the template for n associations. In this regard, the 'Java Code' (J) as a part of the *short template*, must be replaced by the next desire *short template*. Also, the object variable (OV) from the first template must keep the result of that template and pass it to the next template as the input of the class. This means that the result of each template is kept and stored as OV. In the next template one of the classes modifies and receive the values from OV.

5.5.1 Simple Associations

For instance, assume that there are 3 classes: Student Name/Class1, Course/Class2, Marks/Class3 and the programmer would like to find all student names which are "Arash". The next step is to find who does not take Maths and then find who gets a mark above 50 in those other courses. In this case, there is filtering at the

beginning for class1, then an anti-join of class1 and class2 and at the end association of traversal between class2 and class3. The following template for this example is designed by using the above *short templates*:

$\mathrm{T}_{F} < F, C, OV1, < T_{AJ} >> .... \ \mathrm{T}_{F,A\,J} < F, C, < OV1, F_2, C, OV2, J >> .... \ \mathrm{T}_{F,A\,J,T\,A} < F, C, < OV1, F_2, C, JC, < OV2, F_3, C, JC, OV, J >>>$

((SELECT (Student) ANTIJOIN (Course)) JOIN (MARKS))

The actual templates which are presented in Sections 5.2, 5.3 and 5.4 are replaced by the very last *short template* ($\mathrm{T}_{F,A\,J,T\,A} < F, C, < OV, F_1, F_2, C, JC, < OV, F_2, F_3, C, JC, OV, J >>>$), so the final template is created as follow:

```
{
     \\Filtering template\\
     Get OV;
     Query query1 = em.createQuery(
     <SELECT * FROM student s WHERE s.s-name = "Arash" >);
     List list1 = query1.getResultList();
     Iterator iterator1= list1.iterator();
      while(iterator1.hasNext())
       {
          if <CONDITIONS> then
            {
            Update OV to OV1;
            \\RESULT = OV1\\

            \\OV1 is now includes the students from
             \\the 'student' class which have "Arash" name.\\

                 \\Anti join template\\

                 \\Passing OV1 to the new template
                 \\ as the value of Class1\\
                 \\Instead of doing Antijoin between
                 \\student and course classes,
                 \\ the antijoin would happen between the result of
```

```
\\ the previous template
\\ which is already saved in the
\\ object variable 'OV1' and the 'course' class\\

Get OV1;
Query query2 = em.createQuery(
<SELECT ov1 from OV1>);
List list1 = query2.getResultList();
Iterator iterator2= list1.iterator();
while(iterator2.hasNext())
{
Query query3 = em.createQuery(
<SELECT c from course c WHERE
 c.c-name="Math" AND c.course=ov1.c-id=c.c-id>
List list2 = query3.getResultList();
Iterator iterator3= list2.iterator();
If list2.IsEmpty()
   {
   Copy ov1 to OV2;
   \\At this point,
   \\all the students with name "Arash"
   \\whom does not take
   \\course "Math" are selected
  \\ and placed into OV2\\

   \\Join template\\

   \\Passing OV2 to the new template,
   \\ so that the result of antijoin
   \\ 'student' and 'course'
   \\ would be join to the 'marks' class\\


   Get OV2;
   Query query4 = em.createQuery(
   <SELECT ov2 from OV2>);
   List list4 = query4.getResultList();
```

```
                    Iterator iterator4= list1.iterator();
                    while(iterator4.hasNext())
                    {
                        Query query5 = em.createQuery(
                        <SELECT c from course c >
                          WHERE
                           <ov2.c-id= m.m-id AND   m.m-mark > 50>);
                      List list5 = query5.getResultList();
                      Iterator iterator5= list5.iterator();
                      while (iterator5.hasNext())
                       {
                     copy ov2 to OV3;

                     \\Now OV3 include the all the
                     \\students with name "Arash"
                     \\ whom not got "Math" and
                     \\ got mark above 50 in the other courses\\

                     <Java code>; } }
                      }
                }
        }
```

The inner loops results must be calculated and passed to the outer loops for a more complex query.

For instance, if the case is :
FILTERING(FILTERING(A JOIN B) ANTIJOIN (C JOIN D))

then the template for the application must be implemented in the following way:

```
FILTERING {
  ANTIJOIN {
    FILTERING {
      JOIN (A,B) {
       } RESULT OV1
```

```
    } RESULT OV2

    JOIN (C,D) {
    } RESULT OV3
  } RESULT OV4
} RESULT OV5
```

Alternatively, this can be done as follow:

```
JOIN (A,B) {
} RESULT OV1

FILTERING (OV1) {
} RESULT OV2

JOIN (C,D) {
} RESULT OV3

ANTIJOIN (OV2, OV3) {
} RESULT OV4

FILTERING (OV4) {
} RESULT OV5
```

Figure 5.1 illustrates the symbolic tree of the OR application. This tree has three levels. Both join and anti-join patterns are applied to the tree. The patterns must start to apply from the inner nodes. In this tree, each two pairs of nodes will be transferred into one node. Generally this tree presents the process of applying the patterns step by step. Each level includes two nodes and each node represents a SELECT statement which is a part of the OR application. The join template is applied to the nodes on the left side of the main branch. On the right branch the anti-join template is applied.

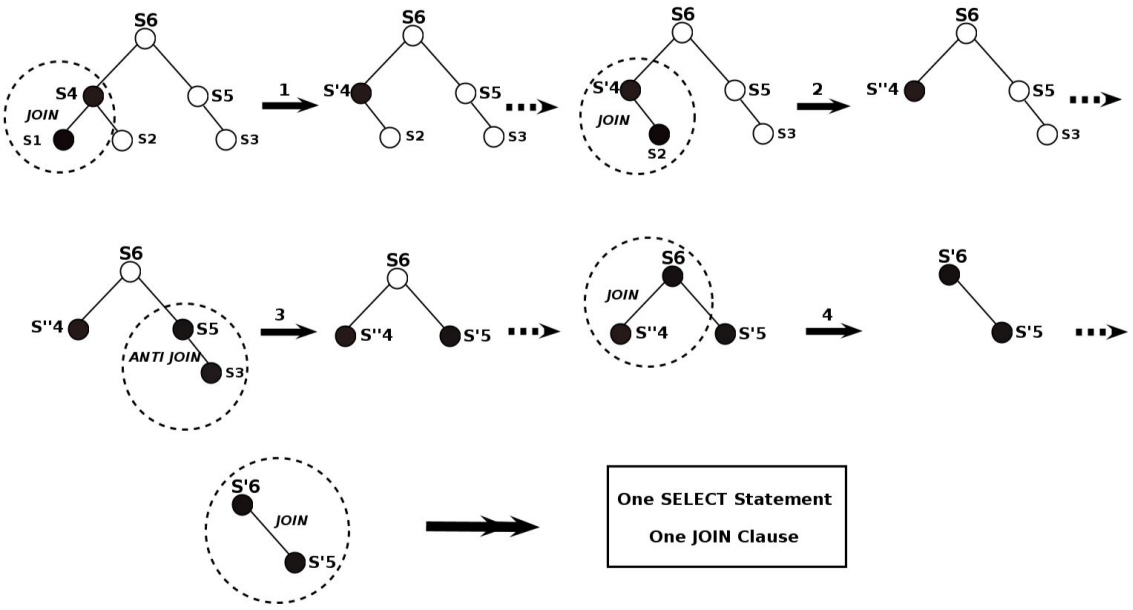

Figure 5.1: Apply patterns symbolically on an application step by step.

Where the outermost SELECT statement is S and all other SELECT statements in the branches are s. Applying the templates has to be started from the two innermost SELECT statements. At the end, there will be only one SELECT statement. This new SELECT statement is used in the join/anti-join clause with the other innermost SELECT statements.

5.5.2 Complex Associations

In this section, a complex query implemented based on the n associations rule.

In this example it is assumed that there are 5 classes: Student/Class1, Enrolment/Class2, Course/Class3, Marks/Class4, Faculty/Class5. The programmer would like to find all the student names who have already enrolled. For this step, a filtering template is used to select all students and a join template is used to find the enrolled students.

The next step is to find who enrolled in 'Science'. Therefore, the result of the previous query needs to join with the course class. For this, the TA template is used.

Now, from the result of previous queries, we are able to find who did not pass that course. Therefore, an antijoin template (AJ) is needed here. Also, we are able to find which faculty the failed students belong to and at the end of the programme count the number of failed students. To implement this part of the query, one join with faculty class and one aggregation template to count the number of students is used.
To implement the above query based on short templates which are presented in section 5.5, the following query needs to be executed:

((((SELECT (Student) JOIN (Enrolment)) JOIN(Course)) AntiJOIN (Marks)) Aggregate (Faculty))

The following short template replaced in the above query should be provided in the correct order.

Short templates:
$\mathrm{T}_F < F, C, OV, J >$
$\mathrm{T}_{TA} < F_1, F_2, C, JC, OV, J >$
$\mathrm{T}_{AJ} < F_1, F_2, C, OV, J >$
$\mathrm{T}_{AG} < A, F, AGC, OV, J >$

Apply short templates:

1. $\mathrm{T}_F < F, C, OV1, < T_J >>$....

2. $\mathrm{T}_{F,T\ A} < F, C, < OV1, F_2, C, , JC, OV2, J >>$....

3. $\mathrm{T}_{F,T\ A,T\ A} < F, C, < OV1, F_2, C, JC, < OV2, F_3, C, JC, OV3, J >>>$....

4. $\mathrm{T}_{F,T\ A,T\ A}, A_J < F, C, < OV1, F_2, C, JC, < OV2, F_3, C, JC, < OV3, F_4, C, OV4, J >>>>$....

5. $\mathrm{T}_{F,T\ A,T\ A}, A_J, A_G < F, C, < OV1, F_2, C, JC, < OV2, F_3, C, JC, < OV3, F_4, C, <$
$OV4, AGC, OV5, J >>>>>$

First the filtering template presented (presented in 1). Then, the join template placed instead of OV of the previous template (presented in 2). Then another join template placed instead of OV2 (presented in 3). Next, to present the anti-join, an anti-join pattern is replaced instead of OV3 (presented in 4). Finally, an aggregation template is replaced instead of OV3. The output of the whole query returned by OV4 (presented in 5). The actual templates which are presented in Section 5.2, 5.3 and 5.4 are replaced in the above short template to create the actual final template of the complex query. So the final template is created as follows:

```
{
     \\Filtering template\\
     Get OV;
     Query query1 = em.createQuery(
     <SELECT * FROM student s>);
     List list1 = query1.getResultList();
     Iterator iterator1= list1.iterator();
      while(iterator1.hasNext())
       {
          if <CONDITIONS> then
            {
            Update OV to OV1;
            \\RESULT = OV1\\

            \\OV1 is now includes the students
             \\ from the 'student'.\\

                 \\Join template\\

                 \\Passing OV1 to the new template
                 \\as the value of Class1\\
                 \\Instead of doing Join between
                 \\student and enrolment classes,
                 \\the Join would happen between
                 \\the result of the previous template
                 \\which is already saved in the
                 \\object variable 'OV1' and the 'enrolment' class\\
```

```
Get OV1;
Query query2 = em.createQuery(
<SELECT ov1 from OV1>);
List list2 = query2.getResultList();
Iterator iterator2= list2.iterator();
while(iterator2.hasNext())
{
Query query3 = em.createQuery(
<SELECT e from enrolment e WHERE
 e.enrolment.status= 'Y'
 AND e.enrolment.id=ov1.s-id >
List list3 = query3.getResultList();
Iterator iterator3= list3.iterator();
If list3.IsEmpty()
   {
  Copy ov1 to OV2;
   \\At this point,
   \\all the students who are
   \\already enrolled are selected
   \\and placed into OV2\\


   \\Second Join template\\

   \\Passing OV2 to the new template,
   \\the result of the first join
   \\'student' and 'enrolment'
   \\would be joined to the 'course' class\\



   Get OV2;
   Query query4 = em.createQuery(
   <SELECT ov2 from OV2>);
   List list4 = query4.getResultList();
   Iterator iterator4= list4.iterator();
```

```
    while(iterator4.hasNext())
    {
        Query query5 = em.createQuery(
        <SELECT c from course c >
          WHERE
          <c.course-name=science AND
             ov2.s-id= c.c-id>);
      List list5 = query5.getResultList();
      Iterator iterator5= list5.iterator();
      while (iterator5.hasNext())
     {
     copy ov2 to OV3;

     \\Now OV3 includes all the
     \\students who are already enrolled
     \\and studying the science course\\



     \\Anti join template\\

\\Passing OV3 to the new template
\\the antijoin would happen between
\\the result of the previous template
\\which is already saved in the
\\object variable 'OV3'
\\ and the 'Mark' class\\

Get OV3;
Query query6 = em.createQuery(
<SELECT ov3 from OV3>);
List list6 = query6.getResultList();
Iterator iterator6= list6.iterator();
while(iterator6.hasNext())
{
Query query7 = em.createQuery(
<SELECT m from marks m WHERE
 m.mark=>50>
```

```
List list7 = query7.getResultList();
Iterator iterator7= list7.iterator();
If list7.IsEmpty()
  {
   Copy ov3 to OV4;
   \\At this point,
   \\all the students
   \\who does not achieve
   \\the pass mark are selected
   \\and placed into OV4\\

   \\Aggregation template\\

   \\Passing OV4 to the new template
   \\the aggregation would happen between
   \\the result of the previous template
   \\which is already saved in the
   \\object variable 'OV4'
   \\and the 'faculty' class\\

   Query query8 = em.createQuery
  ("SELECT ov4 FROM OV4");
  List list8 = query8.getResultList();
  Iterator iterator8= list8.iterator();
  while(iterator8.hasNext())
  {
  Query query9 = em.createQuery
 ("SELECT <count(*)>
  FROM f from faculty f WHERE
  <f.faculty-name=ov4.m-id");
 List list9 = query9.getResultList();
 Iterator iterator9= list9.iterator();
 while (iterator9.hasNext())
 {
  <Java code>; } } } } } } }
 }
}
```

```
}       }
```

Designing a tool which can implement the final template automatically, remains to be determined in future work on this system.

Chapter 6

Experimental Results

The results of the experiments conducted are presented in this section. Real applications are tested for both input and output components of each group of the transformation rules. The total number of *Blocks-Read* operations and also run-time of the applications accessing the different size databases are compared for the first set of experiments. The rest of the experiments compared only the run-time of the applications before and after transformation. The experiments are conducted using the TPC-H Benchmark database which has 300 MB relational data. The Lucid Lynx Ubuntu system running on 3.33 GHz Intel(R), Core(TM)2, Duo CPU with 3.25GB RAM is used to run the applications. In all of the following examples, a class *Supplier* and a class *Lineitem* are used for the experiment. Class *Supplier* consists of 3000 objects while the *Lineitem* class includes objects varies between 400,000 objects to 1,800,000 objects.

The netbeans run-time clock used to measure the run-time of application. The netbeans run-time clock calculates the application's run-time regardless of the network and server traffic by using a byte-count as the basis of its measurement. It measures the amount of data (in bytes) that is processed and/or transmitted. Hence, the measurement is independent to the server load and to the state of the network. A couple of examples were run using both netbeans run-time and wall-clock. The results of the netbeans run-time clock were consistent and reasonable. However, the run-time calculated by the wall-clock were fluctuated and were completely different than the real runtime.

The class *Supplier* and *Lineitem* class were Oracle database tables. Considering this thesis supports the OR application, all the tables are converted to java classes of objects within the Java programming language (Java Persistence API (JPA)). This

conversion is supported by the netbeans environment. The number of objects in each class is considered as the size of the class. Appendix A gives the source code of class *Supplier* and class *Lineitem*.

The Java Persistence API (JPA) is a Java specification for accessing, persisting, and managing data between Java objects / classes and a relational database. JPA was defined as part of the EJB 3.0 specification as a replacement for the EJB 2 CMP Entity Beans specification.

All of the examples were tested in JPA and in the NetBeans IDE 7.1.2 environment. The *Lineitem* and *Supplier* classes in JPA format and also persistence source file are presented in the appendix section.

Figures 6.2 and 6.4 present the average results of *Blocks-Read* for a different number of objects in the Lineitem. The trend line has been incorporated as a polynomial function in order of two for the prediction of the number of blocks read. The trend line (stated as a line of best fit) is a curve that is used to exemplify the behaviour of a set of data to determine if there is a specified pattern. This trend line is an analytical tool on these two-dimensional graphs, to determine the relationship between the number of *Blocks-Read* and number of objects in the Lineitem variables as presented by the equation.

The error bar has also been inserted for each point. As mentioned before the mean value is plotted for reading of blocks which was calculated for each number of objects in the Lineitem by using the AVERAGE function in Excel. In order to reflect the true values, additional information as an error bar is included. There are two ways to describe uncertainty in the presented data. One is with the standard deviation of a single measurement and the other is with the standard deviation of the mean. The second way is known as the standard error and has been incorporated, since the means were shown in the graphs.

First the standard deviation with the STDEV function was calculated. The standard error was also calculated by dividing the standard deviation by the square root of the number of measurements that make up the mean which for this purpose is 4.

Figures 6.1, 6.3, 6.5, 6.6, 6.7 and 6.8, show the average of the execution times for a different number of objects in the Lineitem. The figures show the result of the running optimise and non-optimise applications in different structures. The trend line has been incorporated as a polynomial function in the order of two for the prediction of the execution time for a larger number of objects in the Lineitem. The error bar has also been inserted at each point.

6.1 Motivation Experiment/Join Traversal

Application A is written in Java. This is an example of nested iterations over two classes of objects, called *Lineitem* and *Supplier*. The algorithm count and retrieves all values from *l.sSuppkey* in the *Lineitem* class which has equal values to *s.sSuppkey* in the *Supplier* class. Therefore, joining the classes is required. In this application, *join* operation is designed by nested loops. Each object is compared with an entire class by writing *join* operation with nested loops. Objects which satisfy the *join* condition of two classes are retrieved by running this application. In Application A, a nested loop join iteratively steps through objects from an outer class and reference objects from an inner class join with the current object from the outer class. Each candidate object in the outer class must be iterated before it can be eliminated by the *join* condition with the inner table by writing *join* operation based on nested loops. In Application A, there is an unnecessary overhead to process all of the objects as part of the join process. Application A retrieves the same objects in sSuppkey from both classes as well as the results of counting them. Application A is presented as a nested loop join as follow:

Application A:

```
* @author ZahraDavar(zd991)

package Demo;

import java.math.BigDecimal;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
```

```
import javax.persistence.Persistence;
import javax.persistence.Query;
import java.util.List;
import java.util.Iterator;

public class NewMain1 {
public static void main(String arg[]){
EntityManagerFactory emf=
Persistence.createEntityManagerFactory("newprojectjpa2PU");
EntityManager em=emf.createEntityManager();
try{
EntityTransaction entr=em.getTransaction();
entr.begin();

 Query query1 = em.createQuery
 ("SELECT  s FROM Supplier s", Supplier.class);
    List list1 = query1.getResultList();
    Iterator iterator1= list1.iterator();
    while(iterator1.hasNext())
     {
        Supplier s= (Supplier)iterator1.next();
        Long item = s.getsSuppkey();
        Query query2 = em.createQuery
        ("SELECT  l FROM Lineitem l
            WHERE
              l.sSuppkey=  s.sSuppkey" ,Lineitem.class);
         List list2 = query2.getResultList();
         Iterator iterator2= list2.iterator();
         int counter = 0;
         if (!list2.isEmpty())
           {
              while (iterator2.hasNext())
                {
                   Lineitem l= (Lineitem)iterator2.next();
                   counter ++;
                 }
           }
```

```
            System.out.println( item + "   " + counter);
            }
        }
       finally{
       em.close();
    }
  }
}
```

In order to test the performance of Application A, the total number of *Blocks-Read* operations and the response time was measured for each time running of the application with different size databases. The *Blocks-Read* operations were measured by using Oracle's UTLBSTAT/UTLESTAT scripts to take a snapshot of program activity. It collects instance related performance data. Also, the Netbeans run-time clock was used to measure the run-time.

Experiments started by class *Lineitem* with 400,000 objects and it was gradually increased to 1,800,000 objects. In all of the experiments, size of class *Supplier* was constant at 3000 objects. Figure 6.1 shows the different run-time achieved by running Application A with different size database. The response time starts from approximately 1 hour for 400,000 objects in the class *Lineitem* and increases to almost 6 hours with almost 1,800,000 objects in the class *Lineitem*.

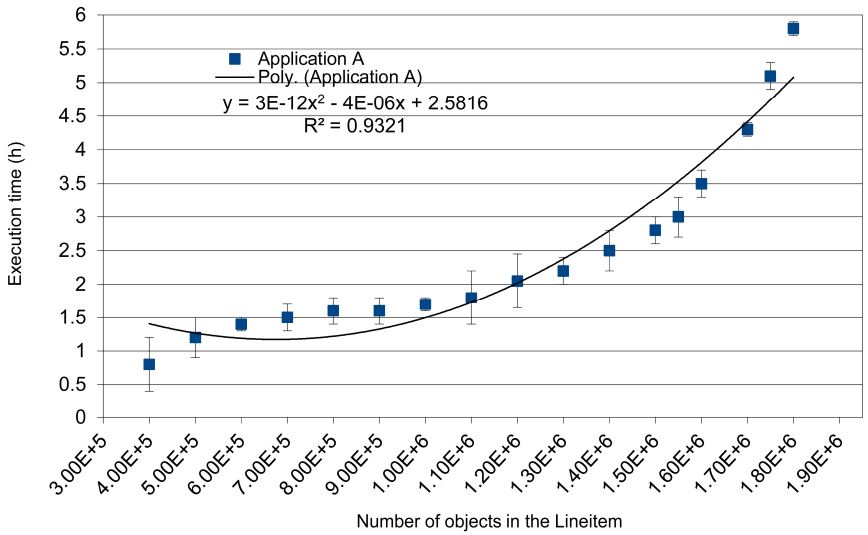

Figure 6.1: Execution Time for Application A

Figure 6.2 illustrates the total number of *Blocks-Read* operations occurring for the performance of the Application A with different sizes of the *Lineitem* class. The number of *Blocks-Read* operations for running Application A started from approximately 92 million blocks for 400,000 objects and ended up with almost 108 million blocks for 1,800,000 objects. The results are shown in the chart in Figure 6.2.

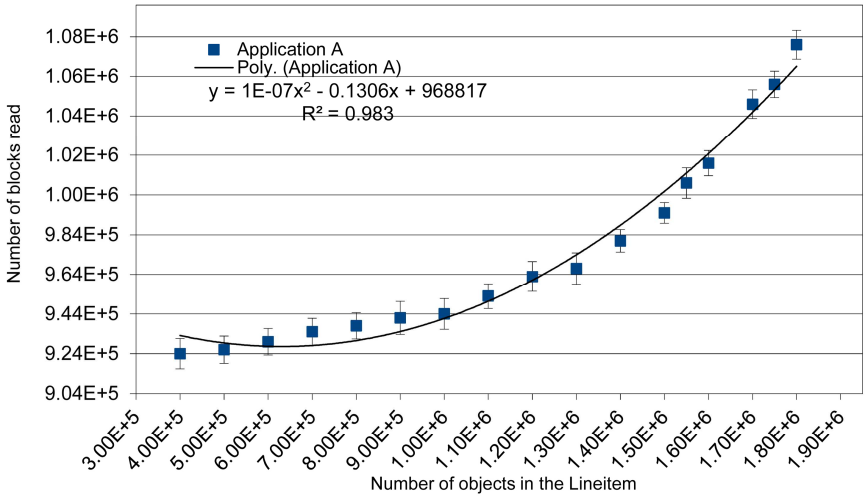

Figure 6.2: Number of *Blocks-Read* for application A

Figure 6.1 and Figure 6.2 show that the increase of the size of the database rises the total number of the *Blocks-Read* operation and run-time of Application A. Also, it can be concluded that the performance problem of this application with a larger database increases exponentially. This implementation leads to read 1,800,000 objects from the database which is a substantial amount of data. Therefore, the join condition is applied to all transferred objects from the server side to the client side. Application A is restructured as shown in Application B to make this application more efficient. Application B has the same output as Application A with a different structure. Application B presents *join* operation with *JOIN* clause as follow:

Application B:

```
* @author ZahraDavar(zd991)

package Demo;

import java.math.BigDecimal;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
```

```
import javax.persistence.Query;
import java.util.List;
import java.util.Iterator;

public class NewMain {
public static void main(String arg[]){
EntityManagerFactory emf=
Persistence.createEntityManagerFactory("newprojectlibraryPU");
EntityManager em=emf.createEntityManager();
try{
EntityTransaction entr=em.getTransaction();
entr.begin();

Query query = (Query) em.createQuery
 ("SELECT l.sSuppkey,
   COUNT(*)
    FROM Lineitem l
     JOIN Supplier s on
      l.sSuppkey = s.sSuppkey", Lineitem.class);

    List list1 = query.getResultList();
    Iterator iterator1= list1.iterator();
    Long ps_suppkey = s.getsSuppkey();

    while(iterator1.hasNext()){
    int counter = 0;
    if (!list1.isEmpty())
 {
          Lineitem l= (Lineitem)iterator1.next();
          counter ++;

 }

        System.out.println( ps_suppkey + "   " + counter);
    }
     finally{
     em.close();
```

```
    }
}
}
```

Application B uses one *SELECT* statement. Application B with more OQL statements, changes the balance of the data-processing. It reduces the total number of objects transferred from the server side to the client side. Therefore, the unnecessary iterations and data transmission are eliminated. With Application A, the total number of *Blocks-Read* operations and the response time for different size *Lineitem* were measured for Application B.

Figure 6.3, illustrates the execution time required to run application B with different sizes of the *Lineitem* class. Application B performed with 400,000 objects in *Lineitem* and increased the objects up to 1,800,000. The results are evidenced in the chart shown as Figure 6.3. The runtime of application B varied between 2 to 4 seconds.
Application B is more efficient than application A. This application requires a few objects to contribute to the final result. Since in Application A, the *join* condition is applied to all objects regardless of the need for them in the final result.

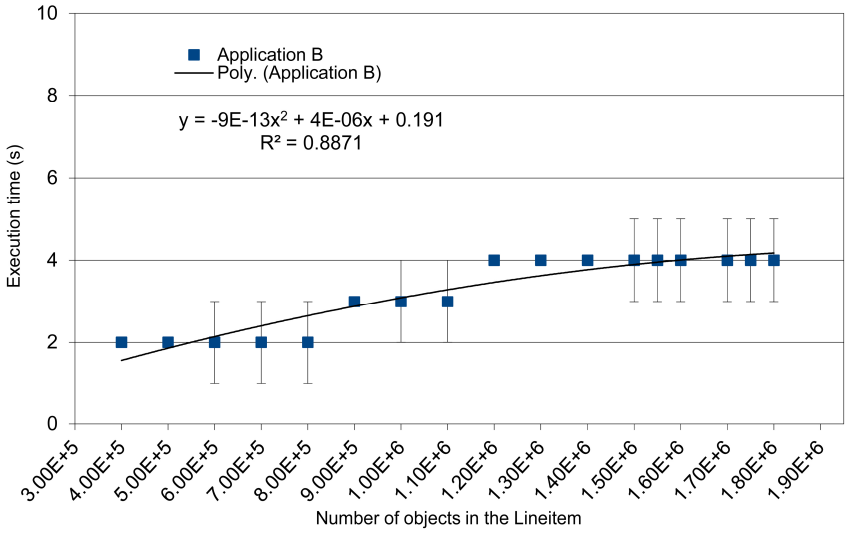

Figure 6.3: Execution Time for Application B

Figure 6.4 illustrates the total number of *Blocks-Read* operation performed by Application B. The number of *Blocks-Read* operations performed by Application B, started from 30,000 blocks for 400,000 objects in the *Lineitem* class and increased to almost 32,000 blocks for 1,800,000 objects. These results show that the increase in the size of the database raises the number of *Blocks-Read* operation. Also comparing Figure 6.4 with Figure 6.2 shows that the number of *Blocks-Read* operation for Application B is significantly less than for Application A.

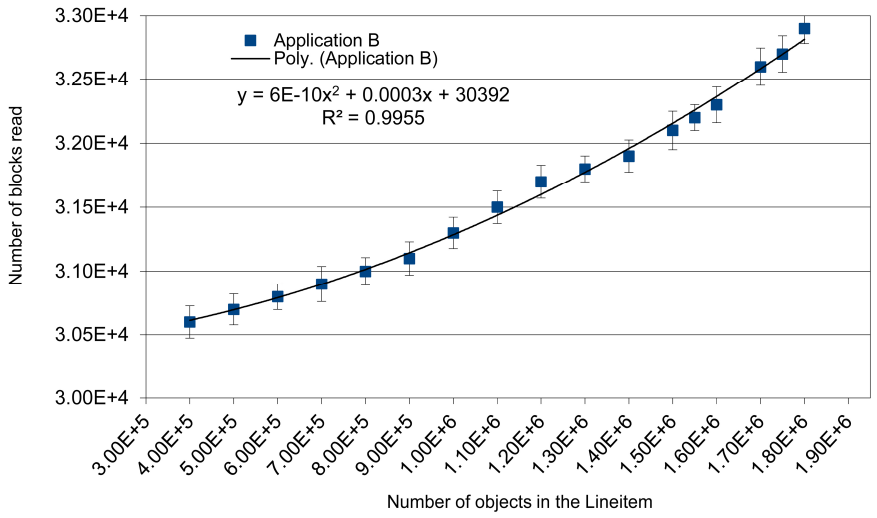

Figure 6.4: Number of *Blocks-Read* for application B

The considerable performance difference between these two implementation is visible by comparing the run-time results of Application A and B which were presented in Figure 6.1 and Figure 6.3. Run-time for Application A exponentially increased from one to six hours, since the same output as Application A is achieved from Application B, with the linear chart and between 2 to almost 4 seconds. To analyse the result, the run-time of both applications for the smallest size and the largest size database were compared. For the *Lineitem* class including 400,000 objects, running Application A took one hour, whereas Application B run in only two seconds. It can be concluded that run-time of Application B, was 1,800 times faster than Application A with 400,000 objects in *Lineitem* class.

For the *Lineitem* class with 1,800,000 objects, Application A took six hours to produce the output while Application B required just four seconds. Application B

was 5400 times faster than Application A with the same size database.

The run-time results of Application A had an exponential growth while run-time of Application B increased linearly, it can be concluded that the performance difference between these two applications are growing exponentially from 400,000 to 1,800,000 objects. Therefore, Application B is much more scalable than Application A.

Also, the number of *blocks read* in Application A was at least 92,000,000 blocks, whereas Application B covered around 30,000 blocks. The significant difference shows that there is no need to read a huge amount of data by restructuring Application A to Application B. Therefore, by comparing all of the results from both applications, it can be concluded that the change in the structure of Application A so that only 300MB data is transmitted to the client side instead of 3GB (1000 times less).

6.2 Anti-Join Traversal

The same classes used in the motivation experiments were employed for running the *anti-join* applications such as *Lineitem* and *Supplier* class. An *anti-join* between two classes returns objects from the first class where there is no similarity in the second class. As all the objects in the Supplier class existed in the Lineitem class, to get an output from the *anti-join* application, first all objects with value of '199' from *Lineitem* class are removed. The following SQL code was performed before running the first *anti-join* application:

"DELETE FROM CSCI315.LINEITEM where L_SUPPKEY=199"

After removing all the objects which had equal value to 199, the maximum size of the *Lineitem* class changed to 1,700,000 objects.
Application C presents an anti-join structure. In Application C, the outer loop receives all the objects from the *Supplier* class. The inner loop collects all the *L_SUPPKEY* from the *Lineitem* class which is matched with *S_SUPPKEY* from the *Supplier* class. Application C, retrieves all values in *L_SUPPKEY* from the *Lineitem* class which do not have the same value in *S_SUPPKEY* from *Supplier* class. By using the default database, the output is 199. Application C is implemented by nested *SELECT* statements.

Application C

```
* @author ZahraDavar(zd991)

package Demo;
import java.math.BigDecimal;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;
import java.util.List;
import java.util.Iterator;

public class NewMain {
public static void main(String arg[]){
EntityManagerFactory emf=
Persistence.createEntityManagerFactory("newprojectjpa2PU");
EntityManager em=emf.createEntityManager();
try{
EntityTransaction entr=em.getTransaction();
entr.begin();

   Query query1 = em.createQuery
 ("SELECT  s FROM Supplier s", Supplier.class);
  List list1 = query1.getResultList();
  Iterator iterator1= list1.iterator();
  System.out.println
 ("Check existence of supplier's data in lineitem class");

 while(iterator1.hasNext())
{

Query query2 = em.createQuery
        (" SELECT  l FROM Lineitem l
```

```
     WHERE l.L_SUPPKEY=" +query1.getInt("s.S_SUPPKEY", Lineitem.class));
List list2 = query2.getResultList();
Iterator iterator2= list2.iterator();
   if (List2.IsEmpty())
    {
       System.out.println(list1.getInt("s.S_SUPPKEY"));
    }

finally{
      em.close();
   }
  }
}
```

The run-time of the application with different sizes of the class *Lineitem* is recorded to test the performance of Application C. One of the outputs of running the above application in Netbeans with almost 1,700,000 objects in *Lineitem* class is as follows:

```
run:
Check existence of supplier's data in lineitem class
ITEM 199not exist in LINEITEM
BUILD SUCCESSFUL (total time: 134 minutes 54 seconds)
```

Figure 6.5 shows the result of running the nested loop structure of the *anti-join* application with different sized *Lineitem* class. The run-time of Application C started from 1 hour for 400,000 objects in the *Lineitem* class and increased to approximately 3 hours for 1,700,000 objects.

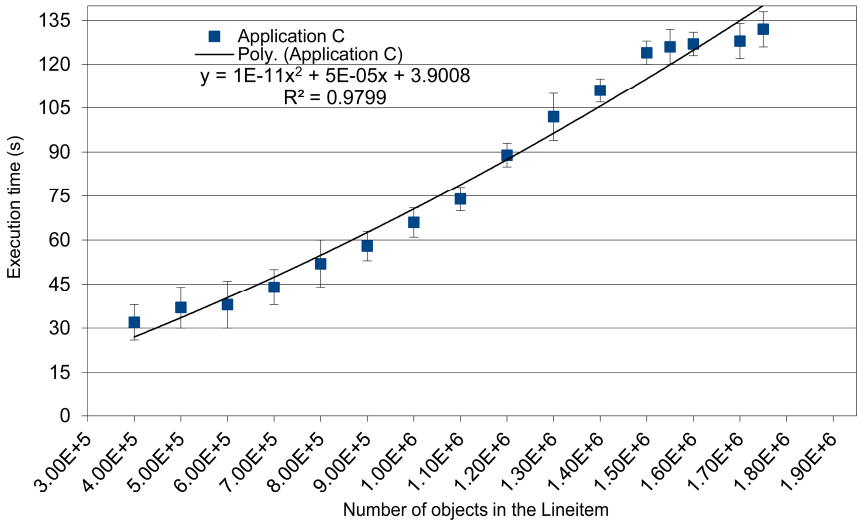

Figure 6.5: Execution Time for Application C

Application D is a restructured version of Application C. Application D implemented *anti-join* by the *left outer join* clause. This application produce the same output as Application C.

Application D:

```
* @author ZahraDavar(zd991)

package Demo;
import java.math.BigDecimal;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;
import java.util.List;
import java.util.Iterator;

public class NewMain {
public static void main(String arg[])
{
EntityManagerFactory emf=
```

```
Persistence.createEntityManagerFactory("newprojectjpa2PU");
EntityManager em=emf.createEntityManager();
try{
EntityTransaction entr=em.getTransaction();
entr.begin();
Query query = (Query) em.createQuery
("SELECT s FROM Supplier s
   LEFT OUTER JOIN Lineitem l WHERE
     s.S_SUPPKEY=l.L_SUPPKEY", Supplier.class);
        List list1 = query.getResultList();
        Iterator iterator1= list1.iterator();
  if (L.SUPPKEY.IsEmpty());
  {
  System.out.println
  ("Checking existence of supplier's objects in lineitem class");

     while(iterator1.hasNext()){
     System.out.println
     ("ITEM " + query.getInt("S_SUPPKEY") + " does not exist in LINEITEM ");
                    }
   }
   }
 finally{
     em.close();
    }
}
}
```

An example of the output:

```
run:
Checking existence of supplier's objects in lineitem class
ITEM 199 does not exist in LINEITEM
BUILD SUCCESSFUL (total time: 3 seconds)
```

Figure 6.6 shows the run-time of application D for different sizes of the *Lineitem* class. The run-time varied between 2 to 4 seconds.

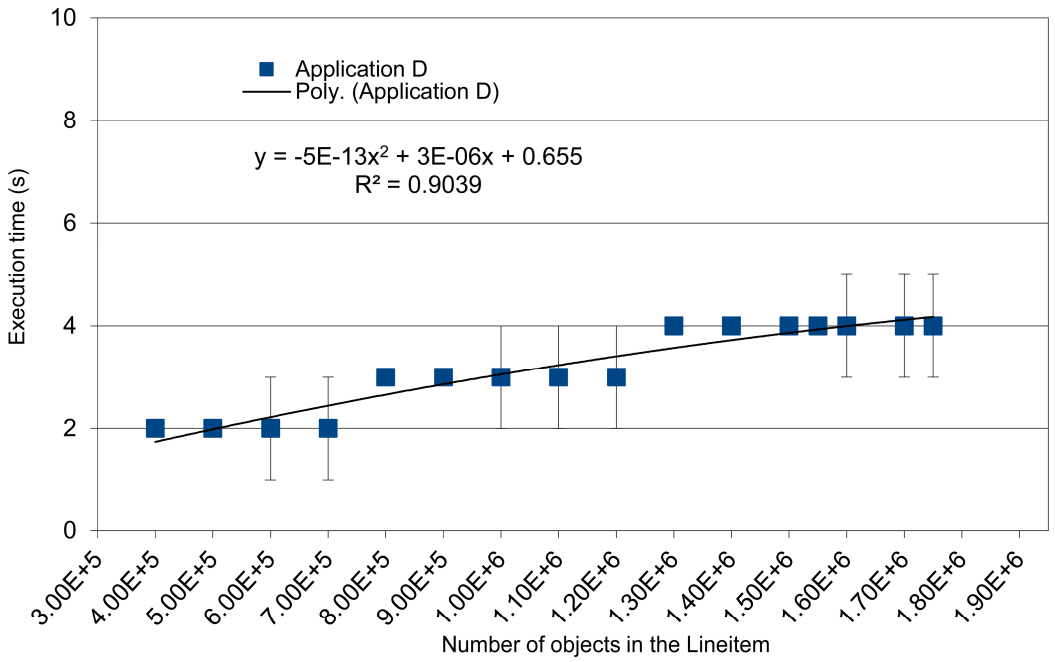

Figure 6.6: Execution Time for Application D

Runtime for Application C and D was 1 hour and 2 seconds respectively for the *Lineitem* class with 400,000 objects. Application D was also more efficient with the database including 1,700,000 objects. The results were released after 3 hours for Application C and after 4 second for Application D.

By having 1,700,000 objects in the database, Application C took around 3 hours to release the output while Application D was run in 4 seconds. Application D ran faster than Application C with the same size database.

The run-time for Application C appears to grow exponentially while the run-time of Application D made a linear chart. Therefore, the performance difference between these two applications appears to be growing exponentially from 400,00 to 1,700,000 objects in *Lineitem* class. As a conclusion, Application D performed more efficiently than Application C. In order to implement an *anti-join* application for a large database with complex objects, the implementation of Application D is more efficient than Application C.

6.3 Aggregation

Real examples were run for counting similar objects from a class to test the performance of aggregation queries. Application E is an aggregation application. It is implemented with nested *SELECT* statements. This application is designed to find the same objects in a class *Lineitem* and count them.

Application E:

```
* @author ZahraDavar(zd991)

package Demo;
import java.math.BigDecimal;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;
import java.util.List;
import java.util.Iterator;

public class NewMain {
public static void main(String arg[]){
EntityManagerFactory emf=
Persistence.createEntityManagerFactory("newprojectjpa2PU");
EntityManager em=emf.createEntityManager();
try{
EntityTransaction entr=em.getTransaction();
entr.begin();

 Query query1 = em.createQuery
 ("SELECT  a.l_Suppkey FROM Lineitem a");
    List list1 = query1.getResultList();
    Iterator iterator1= list1.iterator();
    while(iterator1.hasNext())
     {
         Query query2 = em.createQuery
        ("SELECT  COUNT(*) as total FROM Lineitem b
```

```
            WHERE
              b.l_Suppkey= list1.l_Suppkey " );
          }
          System.out.println("query1.getInt("a.l_Suppkey" )+
                                        query2.getInt("total"));
          }
      finally{
      em.close();
    }
  }
```

Application E is executed several times with a different sized database. The results of running Application E presented in Figure 6.7. Figure 6.7 shows that the run-time of Application E is 1.5 hours for 400,000 objects and increased to around 6 hours for 1,800,000 objects.

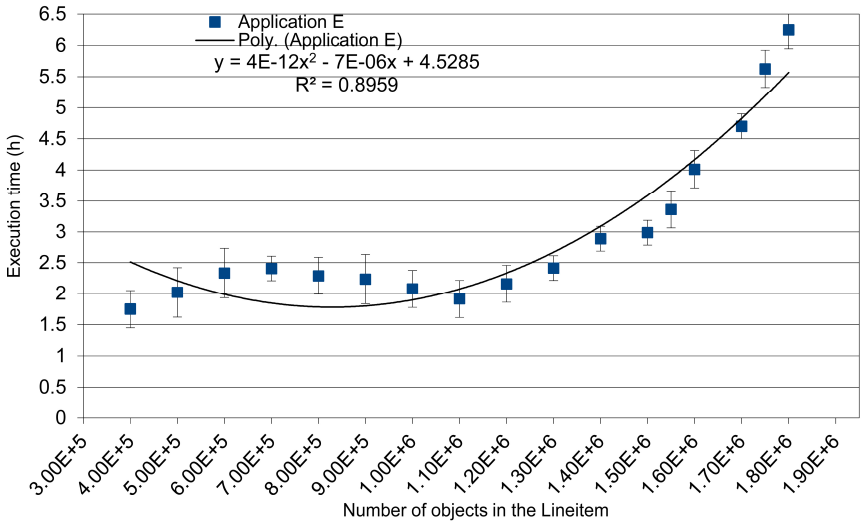

Figure 6.7: Execution Time for Application E

Application F, performs the same operation as Application E using *Group by* clause. The *Group by* clause, grouped the results of counting the same objects and transferring them into the client side. This implementation for counting objects,

eliminates the iteration over all of the objects in the class.

Application F:

```
* @author zd991
package Demo;

import java.math.BigDecimal;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;
import java.util.List;
import java.util.Iterator;

public class NewMain {
public static void main(String arg[]){
EntityManagerFactory emf=
Persistence.createEntityManagerFactory("newprojectlibraryPU");
EntityManager em=emf.createEntityManager();
try{
EntityTransaction entr=em.getTransaction();
entr.begin();

Query query = (Query) em.createQuery
("SELECT l.l_Suppkey, COUNT(l.L_Suppkey) As total
    FROM Lineitem l
      GROUP BY l.l_Suppkey
       ORDER By total");
    List list1 = query1.getResultList();
    Iterator iterator1= list1.iterator();
    while(iterator1.hasNext())
     {
    System.out.println("query.getInt("l.L_Suppkey")"+ query.getInt("total"));
}
```

```
      finally{
      em.close();
     }
}
}
```

The run-time of Application F varied between 3 and 5 seconds. This is presented in Figure 6.8.

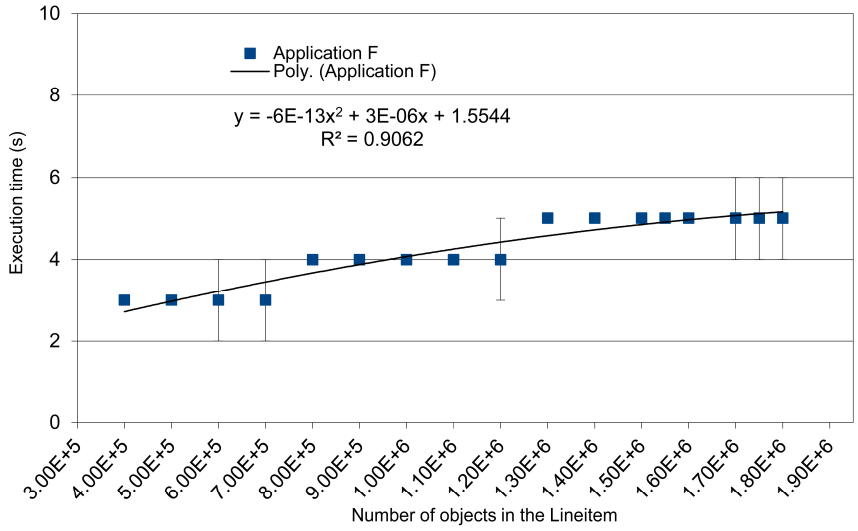

Figure 6.8: Execution Time for Application F

A comparison between Figure 6.7 and Figure 6.8 shows a considerable performance difference between Application E and Application F.

For the *Lineitem* class including 400,000 objects, runtime Application E was 1.5 hours (5400 seconds), whereas for Application F it was 3 seconds. Therefore, Application F was 1800 times faster than Application E for 400,000 objects in the *Lineitem* class.

Application E took 6 hours to release the output while Application F completed in 5 seconds when having 1,800,000 objects in the database. Application F

runs 4320 times faster than Application E when 1,800,000 objects are in the database.

The run-time results of Application E increased exponentially while the run-time of Application F made changes between 3 to 5 second. Therefore, the performance difference between these two applications is growing exponentially from 400,000 objects in *Lineitem* class to 1,800,000 objects. Thus, the implementation of Application F is much more efficient than Application E. The implementation of Application F is recommended for implementing an *anti-join* application for a large database with complex objects.

The results show that by reconfiguring object-relational applications, so that fewer objects are transferred to the client side, more data-processing is performed on the server side which improves the performance of the application.
The same application is applicable for the rest of the aggregation functions such as Min and Max. Therefore, to count the minimum or maximum total value of similar objects, the implementation of Application F is more efficient than Application E.

6.4 Analysis of the results

In all three sets of experiments, the optimised version of the application shows better performance than the original version of the program. To achieve better performance of the application, the implementation of the application is changed, to shift more of the data-processing to the server side and to decrease the amount of data transferring from the server side to the client side.

In all of the experiments, the number of the objects in *Lineitem* class varies between 400,000 objects to 1,800,000 objects. Table 6.1 includes all the experimental results for optimise and non-optimise versions of the applications.

In the Table 6.1, symbol h refers to hour and symbol s refers to seconds. Execution time of the non-optimised versions of join application started from one hour and increased to six hours, while, the execution time of the optimised version of the same application run between two to five seconds.

As explained in Table 6.1 the run time of the non-optimised version of the anti-join application for 400,000 objects in the *Lineitem* class is three hours and it increases to 13 hours for 1,800,000 objects in the *Lineitem* class. The optimised version of the anti-join application run between two to four second.

The execution time of the aggregation application which is listed in Table 6.1, is similar to the join application. The non-optimise version of the application was running between one to six hours while the optimised version takes only three to five seconds.

The performance difference between the optimised and non-optimised application is significant in Table 6.1 for both join and aggregation operations. Anti-join application, provides similar results as join since similar algorithms are used for both of them. Antijon is faster than the join application since the data transmission is smaller.

Table 6.1: Analysis of Results

Number of Objects	Time					
	Join		Anti-Join		Aggregation	
	Non-Opt(h)	Opt(s)	Non-Opt(h)	Opt(s)	Non-Opt(h)	Opt(s)
400,000	0.90	2	3.20	2	0.80	3
500,000	0.94	2	3.40	2	1.20	3
600,000	1.60	3	3.80	2	1.60	3
700,000	1.50	2	4.40	2	1.65	3
800,000	1.70	2	5.20	2	1.70	4
900,000	1.80	3	5.80	2	1.80	4
1,000,000	1.90	3	6.30	2	1.50	4
1,100,000	2.00	3	7.40	3	1.60	4
1,200,000	2.05	4	8.90	3	2.05	5
1,300,000	2.20	4	9.80	3	2.20	4
1,400,000	2.50	3	11.20	3	2.50	5
1,500,000	2.60	4	12.20	4	2.60	4
1,550,000	3.00	4	12.50	4	3.00	5
1,600,000	3.50	4	12.80	4	3.50	5
1,700,000	4.30	4	13.00	4	4.30	5
1,750,000	5.20	4	13.15	4	5.10	5
1,800,000	5.80	4	13.25	4	5.80	5

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis solves the problem of implementing efficient OO applications in the client-server environment. The research also eliminates the performance problem of object-oriented applications within a distributed information system.

In this regard, some experiments were performed to determine the performance problem of object-relational applications with real applications. Based on the experimental results certain transformation rules, have been proposed to shift more data-processing to the server side. This approach increases the amount of non-procedural code as opposed to procedural code in the body of object- oriented applications. Therefore the amount of data transferred from the client side to the server side. Only the necessary objects which satisfy the condition of the application is transferred from the client side to the server side. The proposed transformation rules, results in high performance relational applications.

Three types of transformation rules are considered and their correctness are proven based on the Floyd Hoare logic. Software patterns of the rules are also presented to make the rules more applicable. The patterns are based on JAVA templates.

These applications can be improved with the use of new support tool. A support tool can be designed to apply the patterns automatically to the applications. This support tool can be used by application developers. The tool must divide the application into different parts and match each part with one of the patterns to apply the patterns. As a result the final pattern of the application must be created. This tool must apply the transformation rules on the final pattern and as a result the

optimised version of the application.

Note that the research presented in this thesis focuses on reducing the overall run-time of OO applications in the client-server database environments. This was predominantly achieved by reducing the amount of data that needs to be transmitted and by reducing the client side compute requirement. The proposed approach increases the server side load, though this impact can be neglected in distributed server systems. Since the load can be distributed across servers.

7.2 Future Work

Developing a compiler-level optimizer tool for the rules was not in the scope of this thesis and remains for future work and consideration at PhD level. This optimiser tool needs to recognise the right rule for non-optimised code and automatically translates the code into an equivalent but more efficient application. To implement this tool, one or more modern ORM frameworks (Hibernate, Ruby or LINQ) which were presented in Chapter 2 can be used.

Appendix A

Source code in JPA Format

A.1 Class 'LINEITEM' Source Code

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package Demo;

import java.io.Serializable;
import java.math.BigDecimal;
import java.util.Date;
import javax.persistence.*;
import javax.xml.bind.annotation.XmlRootElement;

/**
 *
 * @author zd991
 */
@Entity
@Table(name = "LINEITEM")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name =
    "Lineitem.findAll", query = "SELECT l FROM Lineitem l"),
    @NamedQuery(name =
    "Lineitem.findByLOrderkey",
```

```
query = "SELECT l FROM Lineitem l WHERE
 l.lineitemPK.lOrderkey = :lOrderkey"),
@NamedQuery(name =
"Lineitem.findByLLinenumber",
query = "SELECT l FROM Lineitem l WHERE
 l.lineitemPK.lLinenumber = :lLinenumber"),
@NamedQuery(name =
"Lineitem.findByLQuantity",
 query = "SELECT l FROM Lineitem l WHERE
  l.lQuantity = :lQuantity"),
@NamedQuery(name =
"Lineitem.findByLExtendedprice",
 query = "SELECT l FROM Lineitem l WHERE
  l.lExtendedprice = :lExtendedprice"),
@NamedQuery(name =
"Lineitem.findByLDiscount",
 query = "SELECT l FROM Lineitem l WHERE
  l.lDiscount = :lDiscount"),
@NamedQuery(name =
"Lineitem.findByLTax",
 query = "SELECT l FROM Lineitem l WHERE
  l.lTax = :lTax"),
@NamedQuery(name =
"Lineitem.findByLReturnflag",
query = "SELECT l FROM Lineitem l WHERE
 l.lReturnflag = :lReturnflag"),
@NamedQuery(name =
 "Lineitem.findByLLinestatus",
  query = "SELECT l FROM Lineitem l WHERE
  l.lLinestatus = :lLinestatus"),
@NamedQuery(name =
 "Lineitem.findByLShipdate",
 query = "SELECT l FROM Lineitem l WHERE
  l.lShipdate = :lShipdate"),
@NamedQuery(name =
"Lineitem.findByLCommitdate",
query = "SELECT l FROM Lineitem l WHERE
```

```
     l.lCommitdate = :lCommitdate"),
    @NamedQuery(name =
    "Lineitem.findByLReceiptdate",
     query = "SELECT l FROM Lineitem l WHERE
     l.lReceiptdate = :lReceiptdate"),
    @NamedQuery(name =
     "Lineitem.findByLShipinstruct",
      query = "SELECT l FROM Lineitem l WHERE
      l.lShipinstruct = :lShipinstruct"),
    @NamedQuery(name =
    "Lineitem.findByLShipmode",
    query = "SELECT l FROM Lineitem l WHERE
    l.lShipmode = :lShipmode"),
    @NamedQuery(name =
    "Lineitem.findByLComment",
     query = "SELECT l FROM Lineitem l WHERE
      l.lComment = :lComment")})
public class Lineitem implements Serializable {
    private static final long serialVersionUID = 1L;



    @EmbeddedId
    protected LineitemPK lineitemPK;
    // @Max(value=?)  @Min(value=?)
    @Basic(optional = false)
    @Column(name = "L_QUANTITY")
    private BigDecimal lQuantity;
    @Basic(optional = false)
    @Column(name = "L_EXTENDEDPRICE")
    private BigDecimal lExtendedprice;
    @Basic(optional = false)
    @Column(name = "L_DISCOUNT")
    private BigDecimal lDiscount;
    @Basic(optional = false)
    @Column(name = "L_TAX")
    private BigDecimal lTax;
```

```
@Basic(optional = false)
@Column(name = "L_RETURNFLAG")
private char lReturnflag;
@Basic(optional = false)
@Column(name = "L_LINESTATUS")
private char lLinestatus;
@Basic(optional = false)
@Column(name = "L_SHIPDATE")
@Temporal(TemporalType.DATE)
private Date lShipdate;
@Basic(optional = false)
@Column(name = "L_COMMITDATE")
@Temporal(TemporalType.DATE)
private Date lCommitdate;
@Basic(optional = false)
@Column(name = "L_RECEIPTDATE")
@Temporal(TemporalType.DATE)
private Date lReceiptdate;
@Basic(optional = false)
@Column(name = "L_SHIPINSTRUCT")
private String lShipinstruct;
@Basic(optional = false)
@Column(name = "L_SHIPMODE")
private String lShipmode;
@Basic(optional = false)
@Column(name = "L_COMMENT")
private String lComment;
@JoinColumns({
    @JoinColumn
    (name = "L_PARTKEY", referencedColumnName = "PS_PARTKEY"),
    @JoinColumn
    (name = "L_SUPPKEY", referencedColumnName = "PS_SUPPKEY")})
@ManyToOne(optional = false)
private Partsupp partsupp;
@JoinColumn
(name = "L_ORDERKEY",
 referencedColumnName = "O_ORDERKEY",
```

```
  insertable = false, updatable = false)
@ManyToOne(optional = false)
private Orders orders;

public Lineitem() {
}

public Lineitem(LineitemPK lineitemPK) {
    this.lineitemPK = lineitemPK;
}

public Lineitem(LineitemPK lineitemPK,
 BigDecimal lQuantity, BigDecimal lExtendedprice,
  BigDecimal lDiscount, BigDecimal lTax,
  char lReturnflag, char lLinestatus,
  Date lShipdate, Date lCommitdate,
  Date lReceiptdate, String lShipinstruct,
   String lShipmode, String lComment) {
    this.lineitemPK = lineitemPK;
    this.lQuantity = lQuantity;
    this.lExtendedprice = lExtendedprice;
    this.lDiscount = lDiscount;
    this.lTax = lTax;
    this.lReturnflag = lReturnflag;
    this.lLinestatus = lLinestatus;
    this.lShipdate = lShipdate;
    this.lCommitdate = lCommitdate;
    this.lReceiptdate = lReceiptdate;
    this.lShipinstruct = lShipinstruct;
    this.lShipmode = lShipmode;
    this.lComment = lComment;
}

public Lineitem(long lOrderkey, long lLinenumber) {
    this.lineitemPK =
     new LineitemPK(lOrderkey, lLinenumber);
}
```

```
public LineitemPK getLineitemPK() {
    return lineitemPK;
}

public void setLineitemPK(LineitemPK lineitemPK) {
    this.lineitemPK = lineitemPK;
}

public BigDecimal getLQuantity() {
    return lQuantity;
}

public void setLQuantity(BigDecimal lQuantity) {
    this.lQuantity = lQuantity;
}

public BigDecimal getLExtendedprice() {
    return lExtendedprice;
}

public void setLExtendedprice
(BigDecimal lExtendedprice) {
    this.lExtendedprice = lExtendedprice;
}

public BigDecimal getLDiscount() {
    return lDiscount;
}

public void setLDiscount(BigDecimal lDiscount) {
    this.lDiscount = lDiscount;
}

public BigDecimal getLTax() {
    return lTax;
}
```

```
public void setLTax(BigDecimal lTax) {
    this.lTax = lTax;
}

public char getLReturnflag() {
    return lReturnflag;
}

public void setLReturnflag(char lReturnflag) {
    this.lReturnflag = lReturnflag;
}

public char getLLinestatus() {
    return lLinestatus;
}

public void setLLinestatus(char lLinestatus) {
    this.lLinestatus = lLinestatus;
}

public Date getLShipdate() {
    return lShipdate;
}

public void setLShipdate(Date lShipdate) {
    this.lShipdate = lShipdate;
}

public Date getLCommitdate() {
    return lCommitdate;
}

public void setLCommitdate(Date lCommitdate) {
    this.lCommitdate = lCommitdate;
}
```

```
public Date getLReceiptdate() {
    return lReceiptdate;
}

public void setLReceiptdate(Date lReceiptdate) {
    this.lReceiptdate = lReceiptdate;
}

public String getLShipinstruct() {
    return lShipinstruct;
}

public void setLShipinstruct(String lShipinstruct) {
    this.lShipinstruct = lShipinstruct;
}

public String getLShipmode() {
    return lShipmode;
}

public void setLShipmode(String lShipmode) {
    this.lShipmode = lShipmode;
}

public String getLComment() {
    return lComment;
}

public void setLComment(String lComment) {
    this.lComment = lComment;
}

public Partsupp getPartsupp() {
    return partsupp;
}

public void setPartsupp(Partsupp partsupp) {
```

```
    this.partsupp = partsupp;
}

public Orders getOrders() {
    return orders;
}

public void setOrders(Orders orders) {
    this.orders = orders;
}

@Override
public int hashCode() {
    int hash = 0;
    hash += (lineitemPK != null ? lineitemPK.hashCode() : 0);
    return hash;
}

@Override
public boolean equals(Object object) {
        if (!(object instanceof Lineitem)) {
        return false;
    }
    Lineitem other = (Lineitem) object;
    if ((this.lineitemPK ==
     null && other.lineitemPK != null)
      || (this.lineitemPK != null &&
      !this.lineitemPK.equals(other.lineitemPK))) {
        return false;
    }
    return true;
}

@Override
public String toString() {
    return "Demo.Lineitem[ lineitemPK=" + lineitemPK + " ]";
}
```

```
}
```

A.2 Class 'SUPPLIER' Source Code

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package Demo;

import java.io.Serializable;
import java.math.BigDecimal;
import java.util.Collection;
import javax.persistence.*;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlTransient;

/**
 *
 * @author zd991
 */
@Entity
@Table(name = "SUPPLIER")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Supplier.findAll",
     query = "SELECT s FROM Supplier s"),
    @NamedQuery(name = "Supplier.findBySSuppkey",
     query = "SELECT s FROM Supplier s WHERE s.sSuppkey = :sSuppkey"),
    @NamedQuery(name = "Supplier.findBySName",
     query = "SELECT s FROM Supplier s WHERE s.sName = :sName"),
    @NamedQuery(name = "Supplier.findBySAddress",
     query = "SELECT s FROM Supplier s WHERE s.sAddress = :sAddress"),
    @NamedQuery(name = "Supplier.findBySPhone",
    query = "SELECT s FROM Supplier s WHERE s.sPhone = :sPhone"),
    @NamedQuery(name = "Supplier.findBySAcctbal",
```

```
     query = "SELECT s FROM Supplier s WHERE s.sAcctbal = :sAcctbal"),
    @NamedQuery(name = "Supplier.findBySComment",
    query = "SELECT s FROM Supplier s WHERE s.sComment = :sComment")})
public class Supplier implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @Basic(optional = false)
    @Column(name = "S_SUPPKEY")
    private Long sSuppkey;
    @Basic(optional = false)
    @Column(name = "S_NAME")
    private String sName;
    @Basic(optional = false)
    @Column(name = "S_ADDRESS")
    private String sAddress;
    @Basic(optional = false)
    @Column(name = "S_PHONE")
    private String sPhone;
    // @Max(value=?)  @Min(value=?)
    @Basic(optional = false)
    @Column(name = "S_ACCTBAL")
    private BigDecimal sAcctbal;
    @Basic(optional = false)
    @Column(name = "S_COMMENT")
    private String sComment;
    @JoinColumn(name = "S_NATIONKEY",
     referencedColumnName = "N_NATIONKEY")
    @ManyToOne(optional = false)
    private Nation sNationkey;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "supplier")
    private Collection<Partsupp> partsuppCollection;

    public Supplier() {
    }

    public Supplier(Long sSuppkey) {
        this.sSuppkey = sSuppkey;
```

```
}

public Supplier(Long sSuppkey, String sName,
 String sAddress, String sPhone,
  BigDecimal sAcctbal, String sComment) {
    this.sSuppkey = sSuppkey;
    this.sName = sName;
    this.sAddress = sAddress;
    this.sPhone = sPhone;
    this.sAcctbal = sAcctbal;
    this.sComment = sComment;
}

public Long getSSuppkey() {
    return sSuppkey;
}

public void setSSuppkey(Long sSuppkey) {
    this.sSuppkey = sSuppkey;
}

public String getSName() {
    return sName;
}

public void setSName(String sName) {
    this.sName = sName;
}

public String getSAddress() {
    return sAddress;
}

public void setSAddress(String sAddress) {
    this.sAddress = sAddress;
}
```

```
public String getSPhone() {
    return sPhone;
}

public void setSPhone(String sPhone) {
    this.sPhone = sPhone;
}

public BigDecimal getSAcctbal() {
    return sAcctbal;
}

public void setSAcctbal(BigDecimal sAcctbal) {
    this.sAcctbal = sAcctbal;
}

public String getSComment() {
    return sComment;
}

public void setSComment(String sComment) {
    this.sComment = sComment;
}

public Nation getSNationkey() {
    return sNationkey;
}

public void setSNationkey(Nation sNationkey) {
    this.sNationkey = sNationkey;
}

@XmlTransient
public Collection<Partsupp> getPartsuppCollection() {
    return partsuppCollection;
}
```

```
    public void setPartsuppCollection
    (Collection<Partsupp> partsuppCollection) {
        this.partsuppCollection = partsuppCollection;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (sSuppkey != null ? sSuppkey.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
            if (!(object instanceof Supplier)) {
            return false;
        }
        Supplier other = (Supplier) object;
        if ((this.sSuppkey ==
         null && other.sSuppkey != null)
          || (this.sSuppkey != null
           && !this.sSuppkey.equals(other.sSuppkey))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return "Demo.Supplier[ sSuppkey=" + sSuppkey + " ]";
    }

}
```

A.3 Persistence File Source Code

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<persistence version="2.0"
xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
   <persistence-unit name="newprojectjpa2PU"
   transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>Demo.Part</class>
    <class>Demo.Orders</class>
    <class>Demo.Supplier</class>
    <class>Demo.Lineitem</class>
    <class>Demo.Region</class>
    <class>Demo.Partsupp</class>
    <class>Demo.Nation</class>
    <class>Demo.Customer</class>
    <properties>
      <property name="eclipselink.id-validation"
       value="NULL"/>
      <property name="javax.persistence.jdbc.url"
       value="jdbc:oracle:thin:@10.9.26.215:1521:jrg"/>
      <property name="javax.persistence.jdbc.password"
       value="*******"/>
      <property name="javax.persistence.jdbc.driver"
       value="oracle.jdbc.OracleDriver"/>
      <property name="javax.persistence.jdbc.user"
       value="csci315"/>
    </properties>
  </persistence-unit>
</persistence>
```

Bibliography

[AA78] Suad Alagic and Michael Arbib. The design of well-structured and correct programs. pages 1–292, 1978.

[ABD+89] Malcolm Atkinson, Francois Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. 8, pages 233–238, USA, Jul 1989. International Conference on Deductive and Object-Oriented Database.

[Aga95] Shailesh Agarwal. Architecting object applications for high performance with relational databases. In *In OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance*. Persistence Software, Inc, 1995.

[BFF+15] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1327–1342, New York, NY, USA, 2015. ACM.

[BK06] Christian Bauer and Gavin King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.

[Blo03] Robin Bloor. The failure of relational database, the rise of object technology and the need for the hybrid database. *Baroudi Bloor International, Inc*, pages 5–6, 2003.

[CB01] Thomas Connolly and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[CSJ+14] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. Detecting performance anti-patterns for applications developed using object-relational mapping. ICSE, pages 1001–1012, New York, NY, USA, Feb 2014. ACM.

[Dat08] Oracle Database. *Object-Relational Developers Guide, 11g, Release 1*, 2008.

[DD88] Joshua Duhl and Craig Damon. A performance comparison of object and relational databases using the sun benchmark. volume 23, pages 153–163, New York, NY, USA, Jan 1988. ACM.

[DG14] Zahra Davar and Janusz Getta. Performance tuning of object-oriented applications in distributed information systems. Number 15, pages 201–208, Lisbon Portugal, Apr 2014. SCITEZPRESS Digital Library Springer Verlag.

[DU04] Suzanne Dietrich and Susan Urban. *An Advanced Course in Database Systems: Beyond Relational Databases*. Prentice-Hall, Inc., NJ, USA, 2004.

[Fin01] Mary A Finn. Fighting impedance mismatch at the database level. *Inter Systems Corporation, InterSystems World Headquarters*, page 2, 2001.

[ICK09] Newton Michael Ireland Christopher, Bowers David and Waugh Kevin. A classification of object-relational impedance mismatch. pages 36–43. IEEE Computer Society, 2009.

[Jac14] Jeff Jacobs. Identifying and refactoring common sql performance anti-patterns. pages 1–9, Northen California, USA, Aug 2014. LLC.

[KA90] Setrag Khoshafian and Razmik Abnous. *Object-Orientation: Concepts, Languages, Databases, User Interfaces*. John Wiley Sons, New Jersey, 1990.

[KB10] Reza Kalantari and Christopher H. Bryant. Comparing the performance of object and object relational database systems on objects of varying complexity. volume 6121 of *Lecture Notes in Computer Science*, pages 72–83, Berlin, Heidelberg, Feb 2010. Springer.

[Kel97] Wolfgang Keller. Mapping objects to tables - a pattern language. In *Proceeding Of European Conference on Pattern Languages of Programming Conference, EuroPLOP*, page Report120/SW1, Germany, 1997.

[LB01] Ramon Lawrence and Ken Barker. Integrating relational database schemas using a standardized dictionary. volume 32, pages 43–48, New York, NY, USA, Mar 2001. ACM symposium on Applied computing.

[LKK00] Sang Ho Lee, Sung Jin Kim, and Won Kim. The bord benchmark for object-relational databases. volume 1873, pages 6–20, London, UK, Jan 2000. Springer.

[Lop04] Andrei Lopatenko. Query answering under exact view assumption in local as view data integration system. pages 14–18, Aug 2004.

[LRMP06] Ivan Lukovi, Sonja Risti, Pavle Mogin, and Jelena Pavievi. Database schema integration process a methodology and aspects of its applying. Novi Sad Journal of Mathematics, 2006.

[LYN+09] Hui Liu, Limei Yang, Zhendong Niu, Zhyi Ma, and Weizhong Shao. Facilitating software refactoring with appropriate resolution order of bad smells. In *Proceedings of the 5th International Workshop on Software Clones*, pages 265–268. ACM, Aug 2009.

[MEW08] Fabrice Marguerie, Steve Eichert, and Jim Wooley. *Linq in Action*. Manning Publications Co., Greenwich, CT, USA, 2008.

[MGB10] Michael Mortensen, Sudipto Ghosh, and James Bieman. Aspect-oriented refactoring of legacy applications: An evaluation. pages 118–140. IEEE, 2010.

[MK98] Ahmed Mostefaoui and Jacques Kouloumdjian. Translating relational queries to object-oriented queries according to odmg-93. volume 1475 of *Lecture Notes in Computer Science*, pages 328–338. Springer, 1998.

[MKF+03] Jan-Eike Michels, Krishna G. Kulkarni, Christopher M. Farrar, Andrew Eisenberg, Nelson Mendona Mattos, and Hugh Darwen. The sql standard. volume 45, pages 30–38, Dec 2003.

[MYK+93] Weiyi Meng, Clement Yu, Won Kim, Gaoming Wang, Tracy Pham, and Son Dao. Construction of a relational front-end for object-oriented database systems. pages 476–483. IEEE Computer Society, Jan 1993.

[Nic07] Lara Nichols. *A Comparison of Object-Relational and Relational Database*. Master of Computer Science Thesis, The Faculty of California Polytechnic University, San Luis Obispo, 2007.

[Nie89] Oscar Nierstrasz. A survey of object-oriented concepts, object-oriented concepts, databases, and applications. pages 3–21. ACM, New York, 1989.

[OLM14] Joseph Ottinger, Jeff Linwood, and Dave Minter. *Beginning Hibernate*. Apress, Berkely, CA, USA, 3rd edition, 2014.

[Ors06] Jaroslav Orsag. Object relational mapping. Bratislava, Slovakia. Diploma Thesis, Comenius University, 2006.

[PRBV90] William Premerlani, James Rumbaugh, Michael R. Blaha, and Thomas A. Varwig. An object-oriented relational database. volume 33, pages 99–109, New York, NY, USA, Nov 1990. ACM.

[RB13] Abhijeet Raipurkar and GR Bamnote. Query processing in distributed database through data distribution. Number 2. IJARCCE, 2013.

[RWD01] Dillon. Tharam Rahayu. Wenny, Chang. Elizabeth and Taniar. David. Performance evaluation of the object-relational transformation methodology. volume 38, pages 265–300. dblp, 2001.

[SBG+10] Tom Schreiber, Simone Bonetti, Torsten Grust, Manuel Mayr, and Jan Rittinger. Thirteen new players in the team: A ferry-based linq to sql provider. *Proc. VLDB Endow.*, pages 1549–1552, Sep 2010.

[SIC98] Son Seungwoo, Yoon Injoong, and Kim Changkap. The technology of object-oriented languages and systems. volume 48, pages 142–147, Washington, DC, USA, Aug 1998. IEEE.

[Sik03] Bagui Sikha. Achievements and weaknesses of object-oriented databases. volume 2, pages 29–41, 2003.

[SKS10] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database system concepts*. McGraw-Hill, New York, 6 edition, 2010.

[Sta98] Frank Stajano. A gentle introduction to relational and object oriented databases. Technical report, ORL, Feb 1998. Technical Report.

[TMJT09] Grust Torsten, Mayr Manuel, Rittinger Jan, and Schreiber Tom. Ferry: Database-supported program execution. Number 9 in SIGMOD, pages 1063–1066, New York, USA, Jul 2009. ACM.

[vZKB06] Pieter van Zyl, Derrick Kourie, and Andrew Boake. Comparing the performance of object databases and orm tools. volume 6, pages 1–11, Republic of South Africa, 2006. South African Institute for Computer Scientists and Information Technologists.

[YZM+95] Clement Yu, Yi Zhang, Weiyi Meng, Won Kim, Gaoming Wang, Tracy Pham, and Son Dao. Translation of object-oriented queries to relational queries. pages 90–97. IEEE Computer Society, Jan 1995.

[ZR11a] Minhaz Zibran and Chanchal Roy. Conflict-aware optimal scheduling of code clone refactoring: A constraint programming approach. In *The 19th International Conference on Program Comprehension*, pages 266–269. IEEE Computer Society, Jun 2011.

[ZR11b] Minhaz Zibran and Chanchal Roy. Towards flexible code clone detection, management, and refactoring in ide. In *Proceedings of the 5th International Workshop on Software Clones*, volume IWSC 11, pages 75–76, New York, NY, USA, Jun 2011. ACM.