Performance Optimisation of Object-Relational Database Applications in Client-Server Environments

Zahra Davar*, Janusz R.Getta†, Handoko‡

School of Computer Science and Software Engineering, University of Wollongong
Email: *zd991@uowmail.edu.au, †jrg.@uow.edu.au
, ‡h629@uowmail.edu.au

Abstract—**The optimisation of object-relational database applications implemented as a mixture of object-oriented and non-procedural code, requires accurate balancing of the data-processing load between the client side and the server side. When there are large amounts of procedural code and less efficient and overly simple algorithms, the majority of the data processing takes place on the client side. As a consequence it usually increases the amounts of data transmitted to the client side and also, the amounts of time needed to process data on the client side. This paper shows how object-relational database applications can be optimised through a better balancing of the data processing load between the client and the server sides. A collection of transformation rules is developed, which replaces the typical iterative structures of procedural code with the equivalent structures of non-procedural code. The software patterns proposed in the paper allow for the automatic optimisation of object-relational applications.**

Keywords–Object-Relational Application; Performance; Transformation Rule; Software Patterns.

I. Introduction

Object-relational mapping and the efficient implementation of object-relational applications, have recently received great deal of attention, especially in commercial environments [1]. Over the past decade, the performance of object-relational applications has become a serious challenge for programmers and database researchers [2].

An object-relational database application is a typical client/server application [3]. In relational systems, the majority of query processing is performed on the server side [4]. Object-relational mapping makes the relational database system available on the server side and visible to an application programmer as a collection of classes of objects on the client side. This means that, relational tables or stored procedures 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 typically implemented in the object-oriented programming language embedded with the simple non-procedural statements of the Object Query Language (OQL).

Programmers, access data on the client side through iterations over the classes of objects or over the results from the processing of OQL statements. Such an approach to 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. For instance, a traversal of an associations between two classes of objects, is implemented as nested loops which iterate over the objects [5]. Programmers typically focus on the logic of an application rather than on how the data will be processed on the client side. This approach to the implementation of an object-relational application, is the main cause of two serious performance problems.

First, the iterations over the large classes of objects on the client side require transferring large amounts of data from the server side. Second, to process these data on the client side, a programmer uses the algorithms which are not as efficient as algorithms which can process the same data on the server side. For example, a traversal of an association on a client side is typically implemented as a join of two relational tables on a server side [5]. Then implementation of *join* operation on the server side with a hash-based or index-based algorithms is much more efficient than implementation of the same *join* operation on the client side with a nested loop algorithm.

Implementing efficient object-relational applications is a serious challenge. There are a number of ways to solve this problem. Recognising the control structures of an application so that it can be rebuilt with more non-procedural code is one solution. This means that, only the objects needed to satisfy the filtering conditions of the application are transferred from the server side to the client side. By using this approach, each relational application written by a programmer can be restructured so as to achieve the same results faster.

This paper, presents a set of transformation rules which can eliminate the iteration over a large number of objects and reduce the amounts of data transmitted over a network by changing the control structures of an application. By applying the rules, some of the procedural components are replaced with OQL statements. This results in faster and more efficient performance of the application. This paper also proposes software patterns which can be used by an application programmer. These patterns, allow the automatic optimisation of object-relational applications.

In the remainder of this paper, experimental results are presented in Section II to show the scale of the problem. Section III reviews the existing research on performance tuning of object-relational applications. The transformation rules are presented in Section IV. Software patterns for different styles of programs are presented in Section V. Section VI contains the conclusion and suggested future work.

II. Case Studies/Experiments

Experiments, were conducted using the TPC-H benchmark database which has 300 MB of relational data. The Lucid Lynx Ubuntu system running on 3.33GHz Intel(R), Core(TM)2, Duo CPU with 3.25GB RAM was used to run the applications. The examples were run in Java Persistence API (JPA) format and in the NetBeans 7 environment. Various experiments were performed for different size databases and the run-time of the applications was measured, using NetBeans run-time clock. In all examples, the *Supplier* class, consists of 3000 objects and the *Lineitem* class varies between 400,000 objects to 1,800,000 objects.

In this section, two sets of experiments are presented to show the motivation of this research. In each set of experiments, the run-time of two different applications with the same output is measured and compared. In each set of experiment, the first application is the original inefficient version of the application and the second one is the improved version. The first set of experiments, presents *anti-join* traversal and the second, illustrates counting from aggregation applications.

A. *Anti-Join Traversal*

The following anti-join application, is used to identify suppliers whose products have never been ordered.

```
{Query query1 = em.createQuery
 ("SELECT  s FROM Supplier s", Supplier.class);
 List list1 = query1.getResultList();
 while(iterator1.hasNext())
{Query query2 = em.createQuery
 (" SELECT  l FROM Lineitem l
   WHERE l.L_SUPPKEY="
    +query1.getInt("s.S_SUPPKEY"));
List list2 = query2.getResultList();
Iterator iterator2= list2.iterator();
 boolean found = false;
 while (iterator2.hasNext())
 if (query1.getInt("s.S_SUPPKEY") ==
   query2.getInt("l.L_SUPPKEY")){
     found=true; } }
    if (!found) {
    System.out.println
    ("ITEM " + query1.getInt("s.S_SUPPKEY")
     + "not exist in LINEITEM ");
    found = false; } }
```

Figure 1. Application C.

To test the performance of Application C, the run-time of the application with different sizes of the class *Lineitem* is recorded.

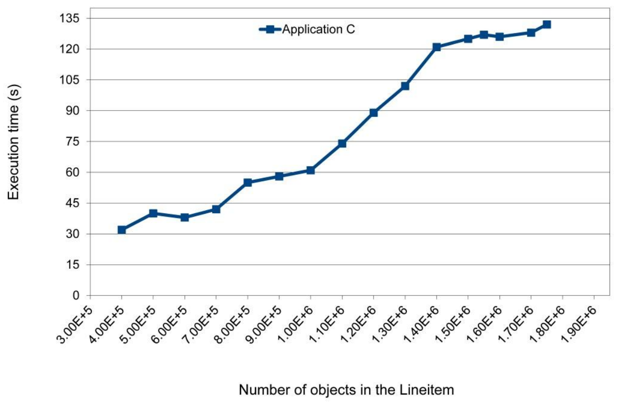

Figure 2. Execution Time for Application C.

Figure 2 shows the result of running the nested loop structure of the *anti-join* application with different size *Lineitem* classes. The run-time of Application C started from 30 seconds with 400,000 objects in the Lineitem class and increased to approximately 130 seconds for 1,700,000 objects.

Next, Application D is implemented using a *left outer join* clause. Using this, only the objects which satisfy the *anti-join* condition transfer to the client side. This *anti-join* application, has the same output as Application C but in the shorter run-time.

```
{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();}
```

Figure 3. Application D.

Figure 4 shows the run-time of Application D for different size *Lineitem* classes. The run-time was varied, between 2-4 seconds.

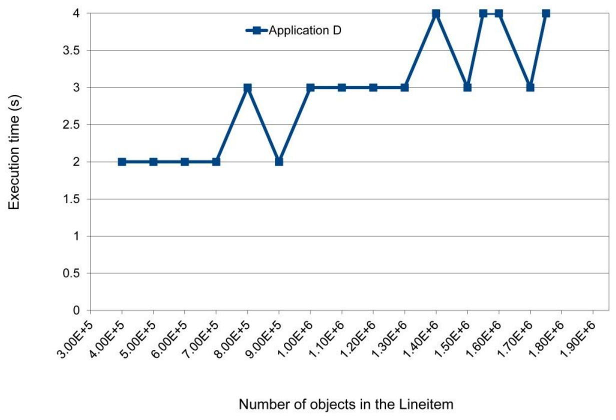

Figure 4. Execution Time for Application D.

Figure 2 and Figure 4 clearly show that Application D is much more efficient than Application C. The run-time for Application C, took 30 seconds with 400,000 objects in the *Lineitem* class and increased to approximately 2 minutes for 1,700,000 objects. Implementing the *anti-join* application with a *left-outer-join*, instead, caused a run-time of between 2 to 4 seconds. On average, Application D ran 26 times faster than Application C. Therefore, to implement an *anti-join* application for large database with complex objects, the implementation of Application D is more efficient than Application C.

B. *Counting Objects*

Two different applications were run for counting objects from a class. Both applications retrieve the same outputs. Application E is implemented with two *SELECT* statements, which iterates on the results of the first *SELECT* statement. This application finds the same objects in *Lineitem* class and counts them. Application E, is implemented by nested *SELECT* statements as follows:

```
{ Query query1 = em.createQuery
 ("SELECT  Distinct l_Suppkey FROM Lineitem l");
    List list1 = query1.getResultList();
    Iterator iterator1= list1.iterator();
    while(iterator1.hasNext())
     { Query query2 = em.createQuery
       ("SELECT  COUNT(*) FROM Lineitem l
         WHERE
           l_Suppkey= list1.l_Suppkey ");
       List list2 = query2.getResultList();
       Iterator iterator2= list2.iterator();}}
```

Figure 5. Application E.

Application E was run several times with different size of databases. The results of running Application E are set-out in Figure 6.

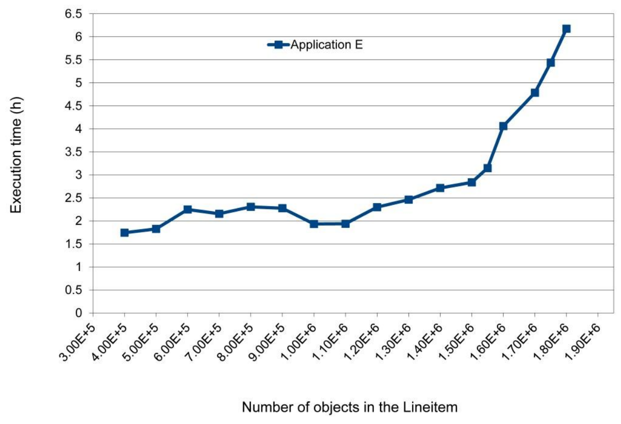

Figure 6. Execution Time for Application E.

Figure 6 shows that, the run-time of Application E took 1.5 hours for 400,000 objects and increased to around 6 hours for 1,800,000 objects.

Application F is implemented by reconfiguring Application E. Application F, used a *Group by* clause to group the results of counting the same objects and transfer them to the client side. This approach, eliminates the necessity for iteration over all of the objects in the class.

```
{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");
     Number countResult=
     (Number) query.getResult();}
```

Figure 7. Application F.

The results of running Application F with different sizes of *Lineitem* class is presented in Figure 8. The run-time of Application F varied between 3 and 5 seconds.

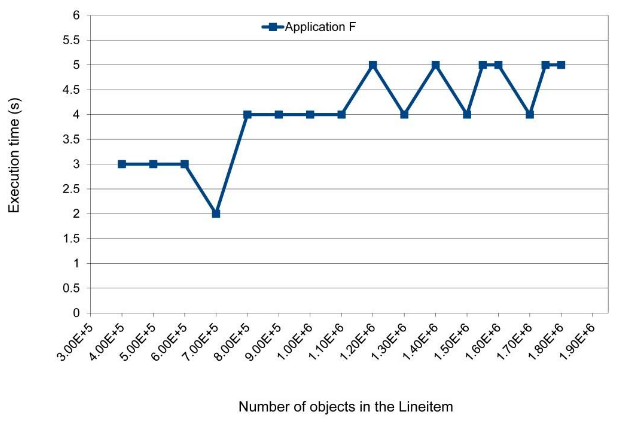

Figure 8. Execution Time for Application F.

A comparison of Figures 6 and 8 shows a very large performance difference between Application E and Application F. Application F was 55 times faster than Application E. The experimental results show that by reconfiguring object-relational applications so that fewer objects are transferred to the client side and more data-processing is done on the server side, better performance of the application is achieved.

III. RELATED WORK

Agarwal [6] proposed the idea of using a client-side object cache in order to increase the performance of the application and 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 in this method, however, is that the complexity of the query must be managed so that it can return instances of commonly used classes with minimum use of joins. In 2006, P. V Zyl et al. focused on comparing the performance of object databases and object-relational mapping tools. This research discussed object-relational mapping in open source applications [7]. This approach, however, only dealt with one framework and was not tried on the distributed or multi-user frameworks which are often used by developers. R. Kalantari et al. compared the performance of object and object-relational database systems. They suggested a number of factors which system developers must consider when selecting a database management system for persisting objects [8] but it was done based on basic query implementation which means that, it did not consider complex queries involving two or more objects. This also means that it is less than optimal for todays applications with complex queries. Rahayu et al. discussed the performance evaluation of object-relational transformation methodology. The aim of this research was to clarify the efficiency of the operations on relational tables based on certain object-relational transformation methodology [9]. The performance of object-relational transformation methodology was also compared with that of the conventional relational model. This work, however, did not involve the dynamic parts of the object orientation. Meng et al. proposed a some transformation rules for object-oriented database systems. The rules used in this research were designed to transform the structural part of an object-oriented database schema into an equivalent relational schema [10]. These rules provided a relational view of the object-oriented database schema for relational users. This research is limited, however, to the structure of a relational front end for object-oriented database systems. The idea of translating queries from an SQL into an OQL in an automatic way, was suggested by Mostefaoui et al. Their method was based on graph representations [11]. A formal approach for translating object-oriented database queries into equivalent relational queries was proposed by Yu et al. who used the same method as Mostefaoui et al. [11][12]. 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 object-relational applications. Grust et al., developed the FERRY language which was designed as an intermediate language which acts as glue that permits a programming style in which developers access database tables using their programming language's own syntax and idiom [13]. In 2010, the same authors extended this approach by proposing the FERRY-based LINQ to SQL approach [14]. Both papers 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 object-relational mapping performance anti-patterns [15], and therefore, improve the systems response time. This is useful but this approach can detect performance bugs and leaves the debugging process for the developer. In our previous work, performance tuning of object-oriented applications in distributed frameworks was discussed. The structure of the proposed approach needs to be upgraded in order to be more efficient. Also, the approach in [16] is only applicable to distributed frameworks and the templates are not general enough to be applicable to complicate applications. It should also be noted that the idea presented in [16], was not sufficiently evaluated through experimental results.

IV. Transformation Rules

The transformation rules presented in this paper convert the non-optimised version of the object-relational database applications into optimised ones in order to provide the necessary efficiency and high speed. The configuration of object-oriented application has been changed by replacing certain procedural parts of the code with non-procedural code. The transformation rules create equivalent applications, where less data is transferred from the server and more data-processing is done on the server side. The transformation rule is applied to the non-optimised version of the program which is an input component and the result is an optimised version of the program, which is an output component. By using more OQL code and changing the structure of the input component, the output component is implemented.

In this paper a filtering (selection) transformation rule, an association anti-traversal rule and an aggregation rule are presented. The rule for *Association Traversal* (presented in our previous work) is presented here, in order to make the JAVA pattern of this specific rule which is obtained in Section V meaningful. Except *Association Traversal* rule, other rules are designed based on the recent experimental results. In the following algorithms, a text *p r o c e s s i n g* means any block of *Java* code.

A. Selection/Filtering Transformation Rule

Each relational application, can include an iteration over one class of objects (selection) which filters the outputs. For this case, the configuration of the application is changed from a program with one *SELECT* statement 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). Therefore, some procedural parts of the code, are replaced with non-procedural code. Figure 9 is the input component algorithm for the filtering rule.

Algorithm 1: Input component
Iteration over one class of objects

```
1 for each t in (SELECT * FROM Class ) do
2     if $\varphi$ $[t.t_1, t.t_2, ..., t.t_n]$ then
3         p r o c e s s i n g
4     end
5 end
```

Figure 9. Input component for Selection Rule.

Figure 10 presents the algorithm of the output component, after applying the above changes to the input component.

Algorithm 2: Output component
Filtering

```
1 for each t in (SELECT * FROM Class WHERE $\varphi$
  $[t_1, t_2, ..., t_n]$ ) do
2     p r o c e s s i n g
3 end
```

Figure 10. Output Component for Selection Rule.

An example of φ $[t_1, t_2, ..., t_n]$ is: Class.Object$_i$=2. Non-relational conditions in the input component φ $[t.t1, t.t2, ..., t.tn]$, will convert to φ $[t_1, t_2, ..., t_n]$ in the output component. This means that, references to the objects $(t.t_1)$, are removed and the output component operates on the name of the properties (t_1). The structure of the entire expression and all contents, however, remain unchanged.

B. Association Anti-Traversal/Anti-Join Transformation Rule

A common input component algorithm, for the *anti-join* rule is introduced in Figure 11. The input component, includes a variable which is *False* by default. This variable will become true if the *SELECT* statement finds any object from class 2, which satisfies the same condition as the object in class 1. *Anti-join* applications, retrieve objects from the second class, which do not exist in the first class.

Algorithm 3: Intput component
Anti-Join by Variable

```
for each t in (SELECT * FROM Class 1) do
    Found = False
    for each s in (SELECT * FROM Class 2 WHERE
    Class 2.Member_j = Class1.t) do
        Found = True
        Exit
    end
    If not Found processing
end
```

Figure 11. Input Component for Association Anti-Traversal.

The algorithm of the output component for association anti-traversal is presented in Figure 12. Our experimental results show that the algorithm presented in Figure 12, can be used as an input component for any possible implementations of the *anti-join*'s input components.

Algorithm 4: Output component
Anti-Join by Left outer join

```
for each p in (SELECT * FROM Class1 Left Outer
Join Class2 on Class2.Member_j = Class1.Member_i)
do
    if Class2.Member_j is Null then
        processing
    end
end
```

Figure 12. Output Component for Association Anti-Traversal.

In our approach, the output component is written by only one *SELECT* statement. The *left-outer-join* is used in the output component of this rule to select the objects with the same condition and makes it unnecessary to transfer them to the client side.

C. Aggregation

Based on our recent experimental results, the input component of the aggregation rule is designed as Algorithm 5. The rule, can support all different types of aggregation applications. The aggregation rule is based on finding similar objects in a class and then applying the aggregation function. For instance, for counting similar objects from a class of objects, F(x) can be a *COUNT(*)* in the input component and *COUNT(Member$_i$)* in the output component. Algorithm 5, is designed using nested *SELECT* statements. In this algorithm, F(x) is the function related to the specific aggregation type, which is used by the application developer. This function can be MIN, MAX, SUM, AVG, or COUNT.

Algorithm 5: Input component
Aggregation with nested loop

```
for each t in (SELECT a.Member_i FROM Class a) do
    for each s in (SELECT F(x) FROM Class b
    WHERE b.Member_i=a.Member_i) do
        processing
    end
end
```

Figure 13. Input Component for Aggregation.

Algorithm 6 presented in Figure 14, used *Group by* clause to group the necessary objects and transfer them to the client side.

Algorithm 6: Output component
Aggregation with grouping objects

```
for each t in (SELECT Member_i, F(x) FROM Class
Group by Member_i) do
    x = getInt(Member_i)
    y = resultset(F(x))
    processing
end
```

Figure 14. Output Component for Aggregation.

By using the *Group by* clause, less objects will be transferred from the server side to the client side. This means that, less run-time is needed to run a application.

D. Iterations over two classes of objects/Association Traversal Rule

Algorithm 7, includes two nested *SELECT* statements which performs the *JOIN* operation. Algorithm 7 is the input component for association traversal applications.

Algorithm 7: Input component
Iterations over two classes of objects

```
for each t in (SELECT * FROM Class1 WHERE $\varphi$
$[t_1, ..., t_n]$) do
    for each s in (SELECT * FROM Class2 WHERE $\gamma$
    $[s_1, s_2, ..., s_n] + \gamma' [<s_1, t_1>, ..., <s_n, t_n>]$) do
        processing
    end
end
```

Figure 15. Input Component for Association Traversal.

In the output component, two *SELECT* statements are merged into one *SELECT* statement with a *JOIN* clause. More non-procedural code is used to write the output component.

Algorithm 8: Output component
Join

```
1 for each p in (SELECT * FROM Class1 Join Class2 on
2 $\gamma$' $[<t_1,s_1>,<t_1,s_2>,...,<t_n,s_n>]$
3 Where
4 $\varphi$ $[t_1,...,t_n]$ || $\gamma$ $[s_1,s_2,...,s_n]$ do
5     p r o c e s s i n g
6 end
```

Figure 16. Output Component for Association Traversal.

An example of the function: γ' $[<s_1,t_1>,<s_2,t_1>,...,<s_n,t_n>]$ in the input component is Class2.Member$_i$ = Class1.Member$_j$. They are the relational conditions of the application. The filtering conditions of Class2 are presented as γ $[s_1,s_2,...,s_n]$. Concatenation in the output component can merge the non-relational conditions of both classes [16]. By using this rule, filtering conditions were applied to the objects on the server side and as a result, only necessary objects which satisfy the *JOIN* condition will transfer to the client side. This means that much data will remain on the server side. This leads to changing the balance of the data-processing between server and client and, as a result, enhances the performance of the application.

V. Software Patterns for Object-Relational Applications

An input component of any transformation rule, is a non-optimised version of an object-relational application. An object-relational application can be written in different ways, and these require a large number of rules to support and optimise them. To use the transformation rules, input components based on what most application programmers use, were needed. To solve this problem, number of software patterns in JAVA programming language have been suggested. The software patterns presented are based on new implementation of the rules and they are all standardised with the JAVA template. As long as the input component, which is the non-optimised version of the application is consistent with the following patterns, then the rules can be applied to the application and optimise it. Depending on the application, the name of the objects, the classes, the functions, the relational conditions and the non-relational conditions will change. Object-oriented programmers need to replace the statement inside $< >$, with the appropriate statement of their own code. The other parts of the pattern remain unchanged.

A. Selection/Filtering Template (SF.Temp)

The input component of the first rule in Section IV-A, must be be consistent with the following template:

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

Figure 17. Selection/Filtering Template (SF.Temp).

CONDITIONS can be any filtering conditions for the class. For instance: Class1.Object$_i$ =X

B. Association Traversal Template (AT.Temp)

If the input component matches this style, the output component of the rule for *iteration over two classes of objects*, which is presented in Section IV-D, is the optimised configuration of the application. In the algorithm, 'n' and 'n-1' are used to show the order of the tables. For instance, if there are two classes of objects, n must be considered as 2 and this means that the outer loop is analysed class 1, while the inner loop is analysed class2:

```
{ Query query'n-1' = em.createQuery
    <SQL SELECT statement from CLASS'n-1'>;
     <GET VARIABLE> ;
  <NON-RELATIONAL CONDITIONS of CLASS'n-1'>;
     List list1 = query'n-1'.getResultList();
     Iterator iterator1= list1.iterator();
     while(iterator1.hasNext()){
       <VARIABLE> =  query'n-1'.getInt(1);
       Query query'n' = em.createQuery
       <SQL SELECT statement from CLASS'n'>
       where <RELATIONAL CONDITIONS>;
        List list2 = query'n'.getResultList();
        Iterator iterator2= list2.iterator();
         while (iterator2.hasNext())
         {
     <NON-RELATIONAL CONDITIONS of CLASS'n'>;
          }
          <JAVA code>;
           }  }
```

Figure 18. Association Traversal Template (AT.Temp).

Assume that, 't' is an object variable which get the objects from the first class and 's' is another object variable which get the objects from the second class. Then: An example of 'NON-RELATIONAL CONDITIONS of CLASS1':
φ $[t.t_1,t.t_2,...,t.t_n]$. Example: Class1.Object$_i$ =X.
An example of 'RELATIONAL CONDITIONS' :
γ' $[<s_1,t_1>,...,<s_n,t_n>]$. Example: Class2.Object$_j$=Class1.Object$_i$.
An example of 'NON-RELATIONAL CONDITIONS of CLASS2' :
γ $[s.s_1,s.s_2,...,s.s_n]$. Example: Class2.Object$_j$=Y.

C. Anti-Join Template (AJ.Temp)

If the input component of the rule, matches this *anti-join* style, then it can be modified according to the rule, which is presented in IV-B.

```
{ 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
       <RELATIONAL CONDITIONS
          between CLASS1 and CLASS2>;
       List list2 = query2.getResultList();
       Iterator iterator2= list2.iterator();
       while (iterator2.hasNext())
        {
        if <ANTI-JOIN CONDITION>{
          VAR = True,
          Exit;
        } }
          if VAR=FALSE
             { <JAVA code>;
             }
        }
}
```

Figure 19. Anti-Join Template (AJ.Temp).

An example of <ANTI-JOIN CONDITION> is: list1.$Member_i$=list2.$Member_j$.

D. Aggregation Template (AG.Temp)

The general template to use the aggregation rule, is presented as below.

```
{ Query query1 = em.createQuery
 ("SELECT <a.Memberi> FROM <CLASS a>");
   List list1 = query1.getResultList();
   Iterator iterator1= list1.iterator();
   while(iterator1.hasNext())
    {
    Query query2 = em.createQuery
    ("SELECT <F(x)>
     FROM <CLASS b>
      WHERE
      <AGGREGATION CONTITION>);
        List list2 = query2.getResultList();
        Iterator iterator2= list2.iterator();
         while (iterator2.hasNext())
        {
       if <AGGREGATION CONTITION> then
         }
         <JAVA code>;
          }  }
```

Figure 20. Aggregation Template (AG.Temp).

To use the output component presented in IV-C, the application must match the following style. F(x) can be any type of the aggregation: MIN , MAX , SUM , AVG or COUNT.

E. n Associations Template

Assume that F: Filtering, C: Condition, J: Java Code, JC: Join Conditions, V: Variable, A: Array, AGC: Aggregation Conditions and OV is an object variable which can keep the results from one template and passes it to the other template. The object variable, takes the results from each template and pass 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 use in the next template. Therefore, generally all the templates can be presented as:

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

For n association, the template will be a mixture of the above templates. To mix the templates, 'J' must be replaced with the desire template. Also, OV from the first template must pass to the next template. The inner most class is considered as class 2. For instance, let us assume that there are 3 classes: Student Name(Class1), Course(Class2), Marks(Class3) and the programmer would like to find all the student names which start with A and then find who does not take Maths and then find who gets a mark above 50 in other courses. In this case, there is a filtering at the beginning for class1, then an antijoin of class1 and class2 and at the end association of traversal between class2 and class3. By using the above short templates, the following template was written for this example:

T$_F$ $< F, C, OV, < T_{AJ} >> $ ——-> T$_{F,A\ J}$ $< F, C, < OV, F_1, F_2, C, OV, J >>$ ——-> T$_{F,A\ J,T\ A}$ $< F, C, < OV, F_1, F_2, C, JC, < OV, F_2, F_3, C, JC, OV, J >>>$

Now by replacing each actual template instead of the name of the template, the final template will design. To make the above example more applicable, the actual templates are replaced in the last achived template (T$_{F,A\ J,T\ A}$ $< F, C, < OV, F_1, F_2, C, JC, < OV, F_2, F_3, C, JC, OV, J >>>$).

```
{\\Filtering template\\
Get OV;
Query query1 = em.createQuery(
<Any SQL SELECT statement
which retrieve objects>);
List list1 = query1.getResultList();
Iterator iterator1= list1.iterator();
while(iterator1.hasNext())
{if <CONDITIONS> then
 {\\Anti join template\\
  Get OV;
  Query query2 = em.createQuery(
  <SQL SELECT statement from CLASS1>);
  GET VAR = FALSE ;
  List list1 = query2.getResultList();
  Iterator iterator2= list1.iterator();
  while(iterator2.hasNext())
  {Query query3 = em.createQuery(
   <SQL SELECT statement from CLASS3>
    where <RELATIONAL CONDITIONS
     of CLASS1 and CLASS2>;
    List list2 = query3.getResultList();
    Iterator iterator3= list2.iterator();
    while (iterator3.hasNext())
    {VAR = True;
    Exit;}}
if VAR=FALSE {\\Join template\\
 Get OV;
 Query query4 = em.createQuery(
 <SQL SELECT statement from CLASS1>);
 <GET VARIABLE> ;
 <NON-RELATIONAL CONDITIONS of CLASS1>;
 List list4 = query4.getResultList();
 Iterator iterator4= list1.iterator();
 while(iterator4.hasNext())
 {<VARIABLE> =  rset4.getInt(4);
 if <CONDITIONS> then{
 Query query5 = em.createQuery(
 <SQL SELECT statement from CLASS2>
 where
  <RELATIONAL CONDITIONS
    of CLASS1 and CLASS2>);
 List list5 = query5.getResultList();
 Iterator iterator5= list5.iterator();
 while (iterator5.hasNext()){
 if <NON-RELATIONAL CONDITIONS
    of CLASS2> then
 <JAVA code>; } } } } } } }
```

Figure 21. n Associations Template.

After obtaining the final design of the input component template, the rules can be applied. To do this, the developers must first find out which styles their application consist of. Then they can use the *n* Associations Template to build their application step by step. Then the rules can be applied to the application and the result is the optimised version of the application. By replacing the short form with the actual template, the final template is achieved as above.

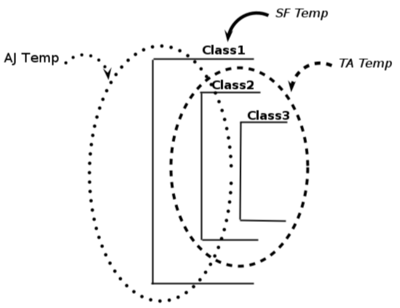

Figure 22. Use templates to prepare the input component

Figure 22, shows how theoretically the templates can be applied to the above example. The templates, however, were applied on the real applications, but the presentation of these, is beyond the scope of this paper.

VI. Conclusion and Future Work

This paper attempted to solve the problem of implementing efficient object-relational applications in client-server frameworks. Certain transformation rules, which can shift more data-processing to the server side, have been proposed. Using this approach, decreases the amount of data transfer from the client side to the server side. Therefore, only the essential objects will be transferred from the server side to the client side. Software patterns of the rules are also presented to make the rules more applicable. The correctness of the rules did not fit in the scale of this paper. Future work will introduce a support tool, which can apply the patterns automatically to the applications.

VII. Acknowledgments

The authors wish to gratefully acknowledge the help of Dr. Madeleine Strong Cincotta in the final language editing of this paper.

References

[1] J. Orsag, ”Object relational mapping”, D. thesis, Comenius University, Bratislava, Slovakia, 2006.

[2] J. Duhl and C. Damon, ”A performance comparison of object and relational databases using the Sun Benchmark”, In Proc. ACM Conference on object-oriented programming systems, languages and applications (OOPSLA ’88), New York:Norman Meyrowitz, 1998, pp. 153-163.

[3] S. Son, I. Yoon, and C. Kim, ”A Component-Based Client/Server Application Development Environment using Java”, In Proc. IEEE Computer Society Conference on the Technology of Object-Oriented Languages and Systems (TOOLS ’98), Washington, 1998, pp. 168.

[4] M. J. Franklin, B. T. Jnsson, and D. Kossmann, ”Performance tradeoffs for client-server query processing”, In Proc. of the ACM SIGMOD international conference on Management of data (SIGMOD ’96), 1996, Jennifer Widom (Ed.), New York, pp. 149-160.

[5] R. Ramakrishnan and J. Database Management Systems. New York: McGraw-Hill, 2002.

[6] S. Agarwal, ”Architecting Object Applications for High Performance with Relational Databases”, In OOPSLA Workshop on Object Database Behavior, Benchmarks, and Performance, Persistence Software, Inc, 1995.

[7] P. van Zyl, D.G. Kourie, and A, ”Comparing the performance of object databases and ORM tools.Boake”, In Proceedings of the annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries (SAICSIT ’06), 2006, Judith Bishop and Derrick Kourie, pp. 1-11.

[8] R. Kalantari and C. H. Bryant, ”Comparing the performance of object and object relational database systems on objects of varying complexity”, In Proceedings of the 27th British national conference on Data Security and Security Data (BNCOD’10), 2010, Lachlan M. MacKinnon (Ed.), Springer-Verlag, Berlin, Heidelberg, pp. 72-83.

[9] J. Wenny Rahayu, E. Chang, T. Dillon, and D. Taniar, ”Performance evaluation of the object-relational transformation methodology,” Data Knowl. Eng, vol. 38, no. 3, pp. 265-300, 2001.

[10] W. Meng, C. T. Yu, W. Kim, G. Wang, T. Pham, and S. Dao, ”Construction of a Relational Front-end for Object-Oriented Database Systems”, In Proc. IEEE Computer Society Conference, 1993, pp. 476-483.

[11] A. Mostefaoui, and J. Kouloumdjian, ”Translating Relational Queries to Object-Oriented Queries According to ODMG-93.”, In: ADBIS, Springer, 1998, pp. 328-338.

[12] C. Yu, Y. Zhang, W. Meng, W. Kim, G. Wang, T. Pham, and S. Dao, ”Translation of Object-Oriented Queries to Relational Queries”, In: Proc. of the 11th Int. Conf. on Data Engineering, 1995, pp. 90-97.

[13] T. Grust, M. Mayr, J. Rittinger, and T. Schreiber ”FERRY: database-supported program execution”, In Proceedings of the ACM SIGMOD International Conference on Management of data (SIGMOD ’09), Carsten Binnig and Benoit Dageville, ACM, New York, 2009, pp. 1063-1066.

[14] T. Schreiber, S. Bonetti, T. Grust, M. Mayr, and J. Rittinger, ”Thirteen new players in the team: a FERRY-based LINQ to SQL provider”, Journal of VLDB, vol. 3, no.2, pp. 1549-1552, 2010.

[15] T. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, ”Detecting performance anti-patterns for applications developed using object-relational mapping”, In Proc. the 36th International Conference on Software Engineering (ICSE 2014), ACM, New York, USA, 2014, pp. 1001-1012.

[16] Zahra Davar, Janusz R Getta, ”Performance Tuning of Object-Oriented Applications in Distributed Information Systems” Presented in 16th International Conference on Enterprise Information Systems (ICEIS 2014), Lisbon, Portugal, 2014.